TSSI

Programación II

Ing. Pinkas Leonardo

Ing. Mariano Kaimakamian Carrau

Ing. Mostovoi Alejandro

Año 2024

(versión 1.6)

Tabla de versiones

Autor	Fecha	Versión	Descripción
Alejandro Mostovoi	12/08/24	1.0	Versión Inicial
Alejandro Mostovoi	26/08/24	1.1	Pruebas unitarias
Alejandro Mostovoi	10/09/24	1.2	Colecciones en Typescript
Alejandro Mostovoi	13/10/24	1.3	Principios S.O.L.I.D
Alejandro Mostovoi	13/10/24	1.4	Estructuras dinámicas
Alejandro Mostovoi	16/10/24	1.5	Correcciones sobre estructuras dinámicas
Alejandro Mostovoi	11/08/25	1.6	Agrega introducción

Índice

Programación Orientada a Objetos **7**
Introducción 7
Elementos de la POO 8
Abstracción 8
Encapsulamiento 8
Modularidad 9
Jerarquía 9
Clases y Objetos 10
Objeto 10
Clase 10
Clase **12**
Generalidades 12
Clase, Tipo, Objeto e Instancia 12
Cuerpo de una clase 12
Visibilidad 13
Funciones, Setters y Getters 14
Ejemplo de de una clase 14
Creación de un objeto 15
Constructor 16
El uso de la palabra this 17
Paquetes 18
Importación de clases 19
Sobrecarga 19
Diferencia entre Typescript y un lenguaje tipado 20
Herencia **22**
Upcasting y Downcasting 23
Sobreescritura 25
El uso de la palabra super 26
Clase abstracta **28**
Composición y agregación **30**
Interfaz **33**
Polimorfismo **35**
Colecciones **35**
Typescript **37**
Array 37
Map 38
Set 39

El uso de la palabra Static **41**
Implicancia de los miembros estáticos 43
Declaración de constantes **44**
Enumerados **45**
Errores y excepciones **46**
Control de excepciones 47
Lanzamiento de excepciones 49
Excepciones personalizadas 50
Principios S.O.L.I.D. **55**
Introducción 55
Importancia de los principios SOLID en el desarrollo de software 56
Principios S.O.L.I.D 56
Single Responsibility (Responsabilidad única) 56
Ejemplo 57
Open for extension, Close to modification (abierto a la extensión, cerrado a la modificación) 58
Ejemplo 58
Liskov Substitution (Sustitución de Liskov) 60
Interface Segregation (Separación de interfaces/conceptos) 60
Dependency Inversion (Inversión de dependencia) 61
Estructuras de datos **64**
Tipo de dato 65
Pilas 66
Definición de la clase pila 66
Operaciones sobre una pila 67
create (crear): 67
isEmpty (está vacía) 67
push (insertar) 67
pop (eliminar) 67
Ejemplo 68
Colas 68
Definición de la clase cola 69
Operaciones sobre una cola 70
create (crear): 70
enqueue (agregar) 70
dequeue (eliminar) 70
isEmpty (está vacía) 71
Ejemplo 71

Listas 71
Definición de la clase lista 72
Operaciones sobre una lista 75
Create (crear): 75
insertFirst (inserta adelante) 75
push (inserta al final) 75
insertOrdered (inserta ordenado) 75
search (buscar un nodo) 75
removeFirst (elimina el primer nodo) 75
pop (elimina el último nodo) 75
delete (elimina un nodo particular) 75
clear (vaciar la lista) 76
SortList (ordenar una lista) 76
insertUnique (inserta sin repetición) 76
Introducción a los patrones de diseño 78
Patrones creacionales 79
Singleton 79
Resumen 79
Factory method 80
Resumen 80
Abstract factory 81
Resumen 82
Builder 82
Resumen 83
Patrones estructurales 84
Adapter 84
Composite 85
Decorator 85
Facade 86
Patrones de comportamiento 87
Strategy 87
Observer 88
Chain of responsibility 88
Template Method 90
Depuración 92
Bug 92
Falla 92
Depuradores 92

Errores fáciles de corregir 92
Errores difíciles de corregir 94
Tipos de Errores 95
Errores Sintácticos 95
Errores Lógicos 95
Errores de Regresión 95
Pruebas **97**
¿Qué es Testing? 97
¿Para qué hacemos Testing? 97
Técnicas de debugging 97
Tipos de testing 98
Tests de caja negra 98
Tests de caja blanca 98
Unitarios 98
De integración 98
De sistema 99
De regresión 99
Testing automatizado 99
Mock Objects 100
¿Para qué usar Mock Objects? 100
Bibliografía **102**

Unidad Temática N°1

~ Programación Orientada a Objetos ~

"La esencia del modelado es la abstracción."

~ Grady Booch

Programación Orientada a Objetos

Introducción

El modelo de objetos abarca los principios de abstracción, encapsulación, modularidad, jerarquía, tipificación, concurrencia y persistencia. Por sí mismos, ninguno de estos principios es nuevo. Lo importante del modelo de objetos es que estos elementos se unen de forma sinérgica.

El análisis y el diseño orientados a objetos requiere una forma diferente de pensar sobre la descomposición (del problema) y produce arquitecturas de software muy distintas respecto del diseño estructurado.

Los métodos de diseño estructurado evolucionaron para guiar a los desarrolladores que intentaban construir sistemas complejos utilizando algoritmos como bloques de construcción fundamentales.

De manera similar, los métodos de diseño orientados a objetos han evolucionado para ayudar a los desarrolladores a explotar el poder expresivo de los lenguajes de programación orientados a objetos, utilizando la clase y el objeto como bloques de construcción básicos.

La razón del éxito generalizado de la programación y diseño orientado a objetos es simplemente que nos ayuda a hacer frente a la complejidad inherente a muchos tipos diferentes de sistemas.

Ahora bien, qué se entiende por programación orientada a objetos (POO)

En líneas generales, podemos decir que la programación orientada a objetos es un paradigma que permite el abordaje de los problemas y búsqueda de soluciones trazando un paralelismo con los objetos que encontramos en la realidad.

De manera más formal, la programación orientada a **objetos** es un método de implementación en el que los programas se organizan como colecciones cooperativas de objetos, cada uno de los cuales representa una instancia de alguna **clase**, y cuyas clases son todas miembros de una **jerarquía** unidas por relaciones de **herencia**.

De esta última definición, podemos destacar los siguientes 3 elementos:

1. La programación orientada a objetos usa objetos, no algoritmos, como sus bloques de construcción lógicos fundamentales.
2. Cada objeto es una instancia de alguna clase.

3. Las clases pueden estar relacionadas entre sí a través de relaciones de herencia.

> *Un programa puede parecer estar orientado a objetos, pero si falta alguno de estos elementos, no es un programa orientado a objetos.*
>
> *Específicamente, la programación sin herencia claramente no está orientada a objetos; eso sería simplemente programar con tipos de datos abstractos.*

Elementos de la POO

Antes de adentrarnos en las definiciones más técnicas del paradigma, es importante analizar algunos conceptos teóricos que sirven como base fundamental para el mismo.

Abstracción

Una abstracción denota las características esenciales de un objeto que lo distinguen de todos los demás tipos de objetos y, por lo tanto, proporcionan límites conceptuales bien definidos, en relación con la perspectiva del espectador (código cliente).

Una abstracción se enfoca en la vista exterior de un objeto y sirve para separar el comportamiento esencial de un objeto de su implementación. En otras palabras la abstracción (como concepto) permite separar “el qué hace un objeto de cómo lo hace“

El concepto de abstracción permite delinear la interfaz de un objeto, lo que nos permite interactuar con el mismo.

La definición de la abstracción (interfaz) de un objeto debe preceder a su implementación.

Encapsulamiento

La abstracción y el encapsulamiento son conceptos complementarios: la **abstracción** se enfoca en el **comportamiento observable de un objeto**, mientras que el **encapsulamiento** se enfoca en la **implementación** que da lugar a este comportamiento.

La encapsulación se logra con mayor frecuencia ocultando información (no solo ocultando datos), que es el proceso de ocultar todos los secretos de un objeto que no contribuyen a sus características esenciales; normalmente, la estructura de un objeto está oculta, así como la implementación de sus métodos.

"Para que la abstracción funcione, la implementación debe ser encapsulada".

Mientras que la abstracción ayuda a las personas a pensar acerca de QUÉ están haciendo, la encapsulación permite que un programa sea modificado con "mínimos" esfuerzos.

En la práctica, esto significa que **cada clase debe tener dos partes**: **una interfaz y una implementación**. La interfaz de una clase captura solo su vista exterior, abarcando nuestra abstracción del comportamiento común a todas las instancias de la clase. La implementación de una clase comprende la representación de la abstracción así como los mecanismos que logran el comportamiento deseado.

La interfaz de una clase es el único lugar donde realizamos todas las suposiciones que un código cliente puede hacer sobre cualquier instancia de la clase (objeto); la implementación encapsula detalles sobre los cuales ningún código cliente puede hacer suposiciones.

En resumen: encapsular es el proceso de compartimentar los elementos de una abstracción que constituyen su estructura y comportamiento; **encapsular sirve para separar la interfaz contractual de una abstracción y su implementación**.

¿Cuál es el beneficio directo de encapsular? Nos da la posibilidad de modificar la representación de una abstracción sin que el código cliente se vea afectado.

Modularidad

Significa agrupar las abstracciones identificadas en esquemas lógicos, que presentan límites bien definidos. Estos límites permiten o ayudan a comprender mejor el funcionamiento de un sistema.

“El acto de dividir un programa en componentes individuales puede reducir su complejidad hasta cierto punto. Aunque particionar un programa es útil por este motivo, una justificación más poderosa para particionar un programa es que crea una serie de límites bien definidos y documentados dentro del programa. Estos límites, o interfaces, son invaluables en la comprensión del programa”

Jerarquía

En todo sistema, salvo en los extremadamente sencillos, existen distintos tipos de abstracciones que podemos identificar. El encapsulamiento ayuda a manejar la complejidad, ocultando los detalles de implementación de las abstracciones identificadas. La modularidad ayuda agrupando de forma lógica las abstracciones identificadas

La jerarquía es un "orden" de abstracciones. Este orden nos permitirá conocer las relaciones jerárquicas existentes en nuestro sistema. Relaciones como "X es un Y" o "W es parte de Z"

Clases y Objetos

Objeto

¿Qué es un objeto?

Desde la perspectiva de la cognición humana, un objeto puede ser cualquiera de estas 3 cosas:

- Algo tangible y/o visible
- Algo que puede ser comprendido intelectualmente
- Algo hacia lo que se dirige el pensamiento o la acción.

Pero en el análisis y diseño orientado a objetos, no sólo nos interesan los objetos reales. Hay objetos que surgen como parte del análisis y/o diseño que colaboran con otros objetos para poder resolver el problema.

Por lo tanto, podemos elaborar una primer definición como la siguiente:

Objeto	Un objeto representa un ítem, individual e identificable, una unidad o entidad, real o abstracta con un rol bien definido dentro del dominio del problema

Pero, si intentamos establecer una definición más formal, lo correcto sería decir que:

Objeto	Un objeto es una entidad que posee un estado, un comportamiento y una identidad.

Clase

Mientras que un objeto es una entidad concreta que existe en tiempo y espacio, una clase representa sólo una abstracción, la esencia de un objeto.

Una clase es una generalización que concentra los aspectos (estado y comportamiento) de los objetos que de ella surjan. Podríamos pensar que una clase es el molde a partir del cual se crean los objetos.

Una clase define las características que tiene que tener todo nuevo objeto que sea creado a partir de ella..

Clase

Generalidades

Una ***clase*** es un archivo de texto en el que se especifican, a nivel general, las características propias de un tipo particular de objeto; en un sentido más práctico se puede pensar a una clase como una plantilla en la que se definen determinados rasgos sin valores predefinidos.

Por ejemplo, se puede crear la clase Puerta definiendo atributos como altura, ancho, profundidad, color y material, lo que permite modelar tanto una puerta de madera como una de metal o de plástico, de cualquier color y de variadas dimensiones; sólo deben asignarse los valores de estos atributos al momento de crear una instancia.

Clase, Tipo, Objeto e Instancia

Aun cuando es bastante habitual intercambiar la palabra ***clase*** por ***objeto***, poseen significados distintos: una ***clase*** es la especificación, es decir, los lineamientos que definen a un ***tipo de objeto***; en cambio un ***objeto*** es una ***instancia***, es decir, un espacio en la memoria RAM que se ha reservado para alojar los valores que el usuario quiera asignarle.

Por cada clase es posible crear tantos objetos como sean necesarios; cada uno requerirá espacio en la memoria RAM para albergar sus valores.

Cuerpo de una clase

Una clase posee un ***nombre*** que la identifica, un nivel de ***visibilidad***, y opcionalmente palabras reservadas que indican si se ***extiende*** de otra clase o si, además, ***implementa*** una interfaz.

También presenta ***atributos***, que son las variables a las que se les asignará valores, y ***funciones***, que son las estructuras que delimitan bloques de instrucciones.

Genéricamente hablando, tanto los atributos como las funciones son conocidos como ***miembros***.

Otro elemento importante es el ***constructor***, que es el método que se invoca cuando se crea una instancia y el lugar en el que todas las asignaciones y acciones que deben

efectuarse al momento de crear un objeto, deben ocurrir; no es obligatorio explicitarlo si no se requiere realizar tarea alguna.

Visibilidad

Todos los miembros poseen ***visibilidad***, que es un indicador sobre el ***nivel de exposición***, tipo de dato al que referencia (en el caso de los atributos) o devuelve (en el caso de las funciones), y un ***nombre*** que los identifica.

Entender el concepto de visibilidad requiere pensar a las clases como elementos herméticos que impiden que sus miembros sean accedidos desde el exterior, imposibilitando que los objetos instanciados de una determinada clase puedan relacionarse con otros.

Para que los objetos puedan interactuar entre sí, es necesario explicitar qué miembros pueden ser vistos desde el exterior, y es allí en donde la visibilidad o nivel de exposición entra en juego.

- ***Privado***: los miembros con este nivel de visibilidad pueden ser referenciados únicamente desde el interior de la clase y no presentan salida al exterior; las subclases no pueden acceder a ninguna función o atributo de su padre. Es el nivel más restrictivo.
- ***Protegido***: es un poco más flexible que el Privado ya que permite que los miembros con este nivel de exposición sean referenciados por clases que se encuentran dentro del mismo paquete, como también por las subclases que extienden de la clase que presenta los miembros protegidos.
- ***Público***: es el menos restrictivo de todos al permitir que cualquier clase pueda referenciar a los miembros con este nivel de visibilidad.

Modificador	Clase	Paquete	Subclase	Mundo
private	✔	❌	❌	❌
protected	✔	✔	✔	❌
public	✔	✔	✔	✔

Funciones, Setters y Getters

Como se mencionara anteriormente, las funciones son agrupadores de instrucciones que tienen como propósito realizar un conjunto de operaciones acotadas, ***encapsulando*** el comportamiento.

El concepto de ***encapsulación*** es muy importante para la programación orientada a objetos ya que permite trabajar conociendo qué hace una función pero no cómo, además de tener la posibilidad -si el código está debidamente estructurado- de reutilizarlo, es decir, invocarlo tantas veces como sea necesario sin tener que repetir código.

Por convención, salvo casos muy particulares, todo atributo debe ser privado y ser accedido mediante setters y getters, que no son más que funciones públicas que permiten la asignación o la lectura del valor de una determinada variable, respectivamente.

De este modo, es posible crear un atributo que sea accedido solamente para la lectura, únicamente para la escritura, o para ambos; de lo contrario, si contara con visibilidad pública, estarían disponibles las dos operaciones.

Ejemplo de de una clase

```
class Person {
  private name: string;
  private lastname: string;

  constructor(name: string, lastname: string) {
       this.name = name;
       this.lastname = lastname;
  }

  getName(): string {
       return this.name;
  }

  setName(name: string): void {
       this.name = name;
  }

  getLastname(): string {
       return this.lastname;
  }
```

```
  setLastname(lastname: string): void {
       this.lastname = lastname;
  }

  sayHi(): void {
       console.log(`Hello, my name is ${this.getFullName()}`);
  }

  private getFullName(): string {
       return `${this.getName()} ${this.getLastname()}`;
  }
}
```

Ejemplo 4.1

Creación de un objeto

Para crear un objeto se emplea la palabra reservada ***new*** seguido del nombre de la clase a instanciar.

Como es habitual en los lenguajes de programación, debe declararse una variable que se corresponda con el tipo de dato de la clase a crear, ya que será allí donde se almacene la referencia del objeto que la instrucción ***new*** provea.

Dicho con otras palabras, la instrucción reservará el espacio para el tipo de objeto en cuestión y devolverá la dirección de memoria, la referencia, que deberá ser almacenada en una variable para poder manipularla posteriormente.

```
myVariable:Person = new Person();
```

Ejemplo 5.1

En el ejemplo expuesto, ***myVariable referencia*** a un objeto del ***tipo Person***, de modo tal que mediante dicha variable ahora ***será posible acceder a cualquiera de los miembros declarados como públicos***.

El acceso a los miembros se realiza a través del uso de un ***punto*** (de allí que reciben el nombre de accessor); en el ejemplo siguiente se muestra la secuencia completa para asignar el nombre de la persona, el apellido e invocar a la función que imprimirá en pantalla el saludo.

```
function main(): void {
  const myVariable = new Person("Al", "Goritmo");
  myVariable.sayHi();
}
```

```
// Invocación de la función main
main();
```

Ejemplo 5.2

Constructor

El uso de la instrucción ***new*** tiene por objetivo reservar el espacio necesario en la memoria RAM para alojar una instancia de la clase que se quiere crear; ejecutando la función que se conoce como ***constructor***.

El constructor está sujeto a las mismas reglas que el resto de las funciones salvo que posee una diferencia muy particular: su firma se caracteriza por la ausencia del tipo de dato que devuelve.

Dado que siempre se ejecuta, es el espacio ideal para incluir instrucciones que sean de carácter obligatorio para la creación de un objeto, como puede ser la inicialización de alguna variable o el cumplimiento de ciertas condiciones.

Si no existe necesidad de realizar operaciones iniciales, puede crearse una clase omitiendo el constructor ya que implícitamente todo objeto requiere de uno y se asume su existencia; en cualquier otro escenario su presencia es requerida y puede definirse con parámetros o sin ellos.

La creación de un objeto haciendo uso del constructor predeterminado se evidencia por los paréntesis vacíos.

```
myVariable:Person = new Person();
```

Ejemplo 6.1

Si fuera obligatorio contar con el nombre y el apellido, por ejemplo, se podría realizar el siguiente ajuste en la clase Person.

```
class Person {
  private name: string;
  private lastname: string;

  constructor(name: string, lastname: string) {
     this.name = name;
     this.lastname = lastname
  }

  // resto de los miembros...
}
```

Ejemplo 6.2

La construcción del objeto nos obligaría a pasar el nombre y el apellido por parámetro.

```
function main(): void {
  const myVariable = new Person("Al", "Goritmo");
  myVariable.sayHi();
}

main();
```

Ejemplo 7.1

Es importante recordar que, si la intención es crear un objeto con determinados valores e impedir que se cambien a posteriori, la clase no debería contar con los setters de los atributos afectados.

El uso de la palabra this

En Typescript, Java, C++ y en muchos otros lenguajes de programación, que implementan el paradigma orientado a objetos, se emplea la palabra reservada ***this*** para hacer ***autorreferencia*** a la instancia (o de la clase si es que se está hablando desde el punto de vista del diseño).

En el ejemplo 6.2 se observa cómo el constructor recibe los parámetros ***name*** y ***lastName*** y almacena los valores en sendas variables homónimas; ante el uso de elementos con el mismo nombre, la palabra reservada ***this*** permite identificar cuál es aquel que pertenece a la clase ya que, de otro modo, el compilador no sabría si la asignación se quiere realizar de los miembros hacia las variables de la función o viceversa.

Podemos destacar que en el caso de Typescript podemos obtener una funcionalidad equivalente, con el siguiente código:

```
class Person {

  constructor(public name: string, public lastname: string) { }

  // resto de los miembros...
}
```

Ejemplo 6.2 bis

Paquetes

Al momento de abordar un proyecto es necesario hacer hincapié en la modularización del mismo ya que esto facilita la legibilidad y mantenibilidad; ambas cuestiones, junto con la reutilización del código, forman la base para un desarrollo veloz.

Uno de los factores que contribuye con esta terna es la organización de las clases en ***paquetes***, que no son ni más ni menos que ***subdirectorios anidados que agrupan archivos bajo un criterio general***.

Dicho esto, sería posible mantener una estructura de paquetes en el que uno de ellos albergue clases utilizadas para modelar datos, otras que posean clases con funciones para validarlos, un tercer paquete destinado a clases que permiten establecer conexiones con servicios web, etc.

La nomenclatura utilizada para el nombre de los paquetes, así como la organización del código fuente debe estar guiada por la necesidad y el juicio del arquitecto encargado de fijar los lineamientos del proyecto.

Importación de clases

Cuando desde una clase se hace referencia a otra que no se encuentra en el mismo paquete, ya sea para instanciarla o para hacer uso de sus miembros públicos estáticos (más adelante se abordará este tema), es preciso indicarle al compilador dónde es que ésta se encuentra.

A los efectos, será suficiente escribir la palabra ***import*** seguido del nombre de la clase a referenciar, indicando de la ruta completa de la clase en cuestión.

```
import Person from “<path/to/PersonClass>”;

function main(): void {
  const myVariable = new Person();
  myVariable.setName("Mariano");
  myVariable.setLastname("Kaimakamian Carrau");
  myVariable.sayHi();
}

// Call the main function to start the program
main();
```

Ejemplo 9.1

Sobrecarga

Del inglés ***overloading***, se trata de una palabra que hace referencia a la capacidad que posee el paradigma orientado a objetos de permitir, dentro de una misma clase, la ***existencia de funciones con la misma firma***; la única condición es que los ***parámetros deben diferir en su tipo de datos, su cantidad y/o ordinalidad de tipos***.

Esta característica permite ordenar el código al emplear un único nombre para funciones que realizan tareas similares pero con parámetros de entrada distintos.

Visibilidad	Tipo	Nombre	Parámetros
public	String	concatPersonInfo	(String name, String lastName)
public	String	concatPersonInfo	(String name, String middleName, String lastName)
public	String	concatPersonInfo	(String name, String lastName, int age)

Ejemplo 10.1

En el ejemplo 10.1 se observa la sobrecarga al existir tres funciones iguales que difieren en cantidad de parámetros y/o tipos, lo que permite invocar a uno u otro método según la cantidad de datos con los que se cuente.

En el ejemplo siguiente se observa un caso que impide la compilación del código ya que se está frente a la existencia de dos funciones de firmas idénticas; el compilador no distingue los nombres de los parámetros sino que analiza la coincidencia basándose en los tipos.

Visibilidad	Tipo	Nombre	Parámetros
public	String	getClientInfo	(int account)
public	String	getClientInfo	(int id)

Ejemplo 10.2

Diferencia entre Typescript y un lenguaje tipado

Como se mencionó previamente, la sobrecarga permite definir el mismo nombre a dos (o más) funciones dentro de la misma clase, siempre y cuando exista diferencia en los parámetros de entrada. Esta diferencia puede ser en el tipo de dato, en la cantidad, en el orden de los mismos o una combinación de todos estos factores.

Ahora bien, este mecanismo funciona a la perfección en un lenguaje (fuertemente) tipado como puede ser Java, C++, C#, etc. Veamos un ejemplo:

```
public class PersonInfo {

  public static String concatPersonInfo(String name, String lastName) {
    return name + " " + lastName;
  }

   public static String concatPersonInfo(String name, String middleName, String
lastName) {
    return name + " " + middleName + " " + lastName;
  }

  public static String concatPersonInfo(String name, String lastName, int age) {
    return name + " " + lastName + ", " + age + " years old";
  }
```

```
  public static void main(String[] args) {
    String fullName = concatPersonInfo("John", "Doe");
    System.out.println(fullName); // Output: John Doe

    String fullMiddleName = concatPersonInfo("John", "Quincy", "Adams");
    System.out.println(fullMiddleName); // Output: John Quincy Adams

    String personWithAge = concatPersonInfo("Jane", "Smith", 30);
    System.out.println(personWithAge); // Output: Jane Smith, 30 years old
  }
}
```

Ejemplo 10.3

Como podemos observar, tenemos 3 métodos con el mismo nombre, al momento de invocar a cada uno de ellos, el compilador puede identificar en base a los parámetros utilizados en dicha invocación a cuál de las 3 versiones del método estamos invocando.

En lenguajes no tipados, o dinámicamente tipados como Typescrypt, Python, PHP, etc. esto no resulta tan sencillo ya que justamente carecemos de tipos de datos. Para lograr el mismo resultado utilizando Typescript, el mecanismo de identificación queda a cargo de quien desarrolle la función. Veamos una posible implementación (hay otras formas de lograrlo):

```
// Dentro de la clase Person

public concatPersonInfo(name: string, lastName: string): string;
public concatPersonInfo(name: string, middleName: string, lastName: string):
string;
public concatPersonInfo(name: string, lastName: string, age: number): string;
public concatPersonInfo(name: string, ...args: (string | number)[]): string {
  if (args.length === 1) {
       return `${name} ${args[0]}`;
  } else if (args.length === 2) {
       return `${name} ${args[0]} ${args[1]}`;
  } else {
       return `${name} ${args[0]}, ${args[2]} years old`;
  }
}

// Usage:
const fullName = concatPersonInfo("John", "Doe");
console.log(fullName); // Output: John Doe

const fullMiddleName = concatPersonInfo("John", "Quincy", "Adams");
console.log(fullMiddleName); // Output: John Quincy Adams

const personWithAge = concatPersonInfo("Jane", "Smith", 30);
console.log(personWithAge); // Output: Jane Smith, 30 years old
```

Ejemplo 10.4

Herencia

La ***herencia*** es el nombre que recibe la característica de la programación orientada a objetos que ***permite crear una clase utilizando a otra existente como base, estableciendo una relación de sucesión en la que la nueva entidad puede hacer uso pleno de los miembros públicos y protegidos de su antecesor, como si fueran suyos***.

Definición	Una clase derivada, posee los atributos y métodos públicos y protegidos de su clase base.

Typescript sólo permite herencia simple, al igual que Java, aunque existen otros lenguajes que admiten heredar de múltiples clases como por ejemplo C++.

Para definir una herencia es requisito utilizar en la firma de la clase la palabra reservada ***extends, seguido del nombre de la clase de la que se quiere extender***.

En los siguientes ejemplos, ***las clases Employee y ProviderEmployee se extienden de Person*** bajo una misma premisa: un empleado es una persona y, por transitividad, posee los mismos atributos que la definen (nombre, segundo nombre, apellido y edad).

```
class Employee extends Person {
  public file: number;
  public laborOld: number;
  public career: string;

  // resto de los miembros... (you can define additional methods/properties here)
}
```

Ejemplo 11.1

```
class ProviderEmployee extends Person {
  public permissionCode: number;
  public companyCode: number;

  // resto de los miembros... (you can define additional methods/properties here)
}
```

Ejemplo 11.2

Cabe destacar que tanto la clase *Employee* como *ProviderEmployee* tendrán los atributos y métodos públicos y protegidos declarados en la clase *Person*.

Habitualmente los problemas suelen requerir modelar elementos que presentan características comunes y estrategias como el uso de la herencia contribuyen a la mantenibilidad y legibilidad del código al evitar que se duplique, además de ordenar conceptualmente las entidades.

Upcasting y Downcasting

La implementación de la herencia abre las puertas al uso del ***upcasting*** y el ***downcasting***, permitiendo que una variable pueda ser tratada como alguno de sus antecesores de los que deriva o bien, cumpliendo ciertos requisitos, que una variable pueda ser tratada como alguno de sus sucesores.

```
class Person {
  sayHi() {
    console.log("hi");
  }
}

class Employee extends Person {
  sayBye() {
    console.log("bye");
  }
}


function main(): void {
  const employee: Person = new Employee();
}

main();
```

Ejemplo 12.1

En el ejemplo anterior se exhibe un ***upcasting*** implícito; nótese cómo un objeto de tipo *Employee* (clase derivada) es asignado a una variable de tipo *Person* (clase base).

Otro modo de realizar el upcasting es mediante el casteo explícito, como se muestra a continuación, en el que se crean instancias de Employee y ProviderEmployee, para luego ser reasignadas a variables del tipo Person.

```
function main(): void {
  const employee: Employee = new Employee();
  const person: Person = (employee as Person)
}

main();
```

Ejemplo 13.1

El uso del ***upcasting*** es importante ya que ***habilita el tratamiento de los objetos de modo más genérico***; por ejemplo, si una empresa quisiera agasajar a todas las personas que contribuyen con su crecimiento y admitiera a los proveedores, todos serían tratados como Person.

Cabe aclarar que ***al momento de hacer un upcasting***, que no necesariamente tiene que remitir a la clase inmediatamente superior, ***se pierde acceso a todos los miembros característicos de la subclase***.

El ***downcasting*** no permite el casteo a otro tipo de dato más que ***al mismo de la instancia original***.

```
function main(): void {
  const employeePerson: Person = new Employee();
  const employee = employeePerson  as Employee;
}

main();
```

Ejemplo 13.2

El ejemplo 13.2 muestra un upcasting implícito y luego un downcasting explícito; esta última acción es posible porque originalmente se trataban de instancias más específicas, lo que implica que al momento de crear los objetos se reservó el espacio necesario para todos sus miembros.

El upcasting no produce una nueva reasignación de memoria, sino que se limita a referenciar al objeto abarcando únicamente los miembros propios de su tipo.

Sobreescritura

Otra característica muy ligada a la herencia de lo que en inglés se conoce como ***overriding***, una práctica mediante la cual un miembro público o protegido declarado en una clase padre, es invalidado para que tome precedencia la de aquella subclase que se quiere instanciar.

De este modo, ***es posible redefinir el comportamiento de las clases que se extienden de otras***, manteniendo el qué hacen pero no el cómo lo hacen, ***lo que resulta en reaccionar de un modo distinto ante un mismo llamado***.

Para conseguir la sobreescritura, simplemente debe reimplementarse el método en la subclase deseada, respetando la firma..

Un ejemplo trivial, pero no por eso menos válido, podría ser el considerar que los empleados de una compañía saludan indicando su nombre completo y número de legajo, mientras que los proveedores requieren indicar su apellido y el id de la compañía a la que pertenecen.

Notar que ***se ha cambiado el nivel de visibilidad del método getFullName() por protected***, ya que la clase Employee requiere el uso del nombre completo; al existir el código que produce ese efecto, no es necesario duplicar código: este ligero cambio es un ejemplo del potencial de la programación orientada a objetos en términos de reutilización.

```
class Person {
  // ... other properties/methods

  public sayHi(): void {
       console.log(`Hello, my name is ${this.getFullName()}`);
  }

  protected getFullName(): string {
       return `${this.getName()} ${this.getLastName()}`;
  }
}
```

Ejemplo 14.1

Los fragmentos de código 15.1 y 15.2 muestran la sobreescritura, realizando los cambios pertinentes.

```
class Employee extends Person {
  // ... employee specific properties/methods

  public sayHi(): void {
         console.log(`Hello, my name is ${this.getFullName()} and my file is
${this.getFile()}`);
  }
}
```

Ejemplo 15.1

```
class ProviderEmployee extends Person {
  // ... provider employee specific properties/methods

  public sayHi(): void {
      console.log(`Hello, my name is ${this.getLastName()} and my company's id is
${this.getCompanyCode()}`);
  }
}
```

Ejemplo 15.1

El resultado final se resume en la existencia de tres clases, emparentadas por la herencia, que ante el llamado de la misma función, producen resultados diferentes.

Es importante tener en cuenta que la ***implementación de la sobreescritura no es obligatoria para todas las subclases***; simplemente es un instrumento más que permite adaptar el código a las necesidades de la solución a través de cambios muy acotados.

Otro detalle fundamental es que, ***aun cuando se haya producido un upcasting, una función sobreescrita se comportará como se haya definido en la subclase debido a la sobreescritura***.

El uso de la palabra super

Los motivos por los que se puede considerar la sobreescritura de un método son:

- ➢ Reemplazar el código
- ➢ Extender funcionalidad
- ➢ Constructores

El primero de los casos es el más trivial y se da cuando el comportamiento de la subclase debe cambiar por completo; el segundo, en cambio, se presenta cuando la

subclase requiere exactamente el mismo algoritmo que el definido en la clase base más un adicional funcional.

Dado que reescribir el código no es una opción viable en términos de mantenibilidad, puesto que duplicarlo no favorece la mantenibilidad, la programación orientada a objetos ofrece una alternativa: utilizar ***la palabra super para hacer referencia a la clase implementadora del miembro al que se quiere acceder***.

Retomando el ejemplo anterior, considerar que la función sayHi() de la clase Person y la sobreescritura en Employee; la única diferencia entre ellas es que la última tiene un mensaje personalizado que varía ligeramente con respecto a la primera.

Conociendo la existencia de la palabra reservada ***super***, ahora es posible reescribir el código para hacerlo más prolijo.

```
class Employee extends Person {
  // ... employee specific properties/methods

  public sayHi(): void {
       super.sayHi(); // Call the base class sayHi method
       console.log(`My file is ${this.getFile()}`);
  }
}
```

Ejemplo 16.1

El efecto de este cambio resultará en la ejecución de la función sayHi() de la clase padre, para luego continuar con la línea de la clase Employee.

Es importante advertir que la llamada a un miembro mediante la palabra reservada ***super*** no requiere necesariamente que se realice al inicio de la función; no existe restricción alguna más que la necesidad de lo que se desea realizar.

El tercer caso resulta mandatorio cuando la clase padre presenta al menos un constructor personalizado, ya que la subclase siempre debe ser capaz de llamar a alguno de los constructores de su padre.

Considerar el siguiente caso:

```
class Person {
  private name: string;
  private age: number;

  constructor(name: string)
  constructor(name: string, age?: number) {
       this.name = name;
       if(age){
         this.age = age;
       }
  }

}
```

Ejemplo 17.1

La clase Person, presenta dos constructores.

La clase Employee está obligada a implementar al menos una llamada al constructor que posee su clase base, en su defecto, un constructor que ejecute super(name) o super(name, age).

```
class Employee extends Person {

  constructor(name: string) {
      super(name);
  }

}
```

Ejemplo 17.2

Clase abstracta

Las clases que se denominan ***abstractas*** difieren de las concretas por presentar dos particularidades, siendo una de ellas la ***imposibilidad de ser instanciada*** y, la otra, que ***puede contener funciones con implementación o sin ellas*** (abstractas).

Su existencia obedece a la necesidad de poder contar con una entidad que ***reúna un conjunto de características necesarias para las subclases, pero no suficientes como para que esta clase en cuestión posea identidad propia (ser instanciada)***, limitando dicha responsabilidad a las subclases que de ella se extiendan.

Por ejemplo, si se requiriese desarrollar una solución para una cadena de estacionamientos, podría ser requisito modelar autos, camiones, bicicletas y motos,

concentrando la mayor cantidad de características comunes en una clase base llamada Vehiculo.

Por su naturaleza no tendría sentido poder crear un objeto Vehiculo ya que es una representación muy genérica en comparación a las subclases, de modo tal que sería candidata para definirla como abstracta y evitar, de este modo, que se instancie inadvertidamente.

Definir si una clase amerita ser abstracta o no, depende de las necesidades del proyecto y la experiencia de quien diseñe la solución.

```
abstract class Vehicle {
  private engine: Engine;

  public getEngine(): Engine {
       return this.engine;
  }

  public setEngine(engine: Engine): void {
       this.engine = engine;
  }

  public abstract turnEngineOn(): boolean;
  public abstract turnEngineOff(): boolean;

  public honk(): void {
       console.log("Tu-tuuuuu");  }
}
```

Ejemplo 19.1

El ejemplo anterior muestra la definición de una clase abstracta, con la declaración de las firmas de dos funciones abstractas y una tercera concreta.

Las funciones abstractas deben ser implementadas por las subclases, para lo cual deben sobreescribirse.

```
class Car extends Vehicle {

  public turnEngineOn(): boolean {
       // Implementación de la función...
       // Por ejemplo:
       console.log("Encendiendo el motor del coche...");
       // ... lógica para encender el motor ...
       return true; // O false si falla el encendido
  }
```

```
  public turnEngineOff(): boolean {
       // Implementación de la función...
       // Por ejemplo:
       console.log("Apagando el motor del coche...");
       // ... lógica para apagar el motor ...
       return true; // O false si falla el apagado
  }
}
```

Ejemplo 20.1

Composición y agregación

Genéricamente hablando, la composición es la relación de dependencia que se establece entre dos objetos, en el que uno es el compuesto y el otro es el componente.

Con un poco más de formalidad, la composición se presenta cuando ***una clase posee al menos un atributo cuyo tipo de dato se corresponde con el de otra clase y no con el de un tipo primitivo***.

Existen dos tipos de relaciones:

- ***Débil (agregación)***: el objeto principal no requiere necesariamente de la existencia del componente.
- ***Fuerte (composición)***: el objeto principal no puede existir sin el componente.

Por ejemplo, construir un auto requiere entre otras cosas de la existencia de un motor y de un aire acondicionado; mientras la relación entre el auto y el motor es fuerte (composición) porque el auto no puede funcionar sin él, la relación entre el auto y aire acondicionado es débil (agregación) porque no es esencial para prestar el servicio.

A continuación, se exhiben las clases que podrían adecuarse al ejemplo.

Notar que ***AirConditioner*** y ***Engine*** presentan miembros ***turnOn()*** y ***turnOff()*** pero sin ningún motivo en particular; en este caso coincide el hecho de necesitar una función para el encendido y apagado.

```
class AirConditioner {
  // ... otros miembros

  public turnOn(): boolean {
```

```
      // Implementación de la función...
      // Por ejemplo:
      console.log("Encendiendo el aire acondicionado...");
      // ... lógica para encender el aire acondicionado ...
      return true; // O false si falla el encendido
  }

  public turnOff(): boolean {
      // Implementación de la función...
      // Por ejemplo:
      console.log("Apagando el aire acondicionado...");
      // ... lógica para apagar el aire acondicionado ...
      return true; // O false si falla el apagado
  }
}
```

Ejemplo 21.1

```
class Engine {
  // ... otros miembros

  public turnOn(): boolean {
      // Implementación de la función...
      console.log("Encendiendo el motor...");
      // ... lógica para encender el motor ...
      return true; // O false si falla el encendido
  }

  public turnOff(): boolean {
      // Implementación de la función...
      console.log("Apagando el motor...");
      // ... lógica para apagar el motor ...
      return true; // O false si falla el apagado
  }

  public speedUp(increase: number): void {
      // Implementación de la función...
      console.log("Aumentando la velocidad en", increase, "unidades.");
      // ... lógica para aumentar la velocidad del motor ...
  }
}
```

Ejemplo 21.2

Habiendo definido los componentes, en el que el primero es débil por ser un “agregado” y el segundo fuerte por el carácter esencial, resta el compuesto.

```
abstract class Vehicle {
  private engine: Engine;
  private airConditioner: AirConditioner;

  constructor(engine: Engine, airConditioner: AirConditioner) {
       this.engine = engine;
       this.airConditioner = airConditioner;
  }

  public turnEngineOn(): boolean {
       return this.engine.turnOn();
  }

  public turnEngineOff(): boolean {
       return this.engine.turnOff();
  }

  public speedUp(): void {
       this.engine.speedUp();
  }

  public turnAcOn(): boolean {
       return this.airConditioner.turnOn();
  }

  public turnAcOff(): boolean {
       return this.airConditioner.turnOff();
  }

  // ... otros miembros
}
```

Ejemplo 22.1

Salvo casos muy particulares, toda composición y agregación implica el uso de algunos de los miembros de sus componentes, de allí que en el ejemplo se observa cómo Vehicle invoca a las funciones de sus componentes: encendido, apagado y aceleración.

Otro detalle importante a mencionar es que suele abusarse del término composición para referirse a la construcción de objetos en base a otros y no tanto al grado de dependencia.

La ***composición*** y la ***agregación*** son esenciales porque no sólo permiten la reutilización del código, sino que ***aportan valor agregado al polimorfismo ya que un objeto***

compuesto, en tiempo de ejecución, puede variar su comportamiento mediante la reasignación de sus componentes.

Interfaz

La identidad, los atributos y las funciones, son los tres elementos que definen a un objeto; en particular, la última de las características mencionadas es una de las más importantes ya que es la responsable del cambio de estado de los mismos; de allí que existen diversas estrategias centradas en torno al comportamiento: sobrecarga, sobreescritura, composición y la herencia (esta última como instrumento que favorece la adopción de comportamiento a las subclases).

Un cuarto recurso es lo que se conoce como ***interfaz.*** **Una interfaz** ***define las firmas de las funciones que debe implementar una clase cuando decide ajustarse a ella***.

Mientras una clase, sea concreta o abstracta, define la fisonomía de los objetos en términos de atributos y comportamientos, una ***interfaz especifica el comportamiento de objetos que pueden o no formar parte de la misma jerarquía***.

Retomando el ejemplo de la composición, ***tanto el aire acondicionado como el motor presentan las funciones turnOn() y turnOff()***; sin embargo, pese a que ***se tratan de objetos de distintas clases,*** desde el punto de vista del comportamiento los dos pueden encenderse y apagarse, lo que permite que ***ambos sean tratados como elementos que sean "encendibles" y "apagables"***.

Las interfaces únicamente definen las firmas de los métodos a ser implementados por las clases.

Continuando con el ejemplo, es posible pensar en una interfaz llamada Turnable con la siguiente especificación de comportamiento.

```
interface Turnable {

  turnOn(): boolean;
  turnOff(): boolean
}
```

Ejemplo 23.1

Las clases que implementan una interfaz deben hacer uso de la palabra ***implements*** seguido del nombre de la interfaz; en caso de implementar más de una, los nombres deben separarse con comas. ***Adicionalmente***, y al igual que ocurre con la sobreescritura convencional, ***las funciones deben sobreescribirse***.

```
interface Turnable {
  turnOn(): boolean;
  turnOff(): boolean;
}

class AirConditioner implements Turnable {
  // ... otros miembros

  turnOn(): boolean {
       // ... implementación de la función...
       console.log('Encendiendo el aire acondicionado');
       return true;
  }

  turnOff(): boolean {
       // ... implementación de la función...
       console.log('Apagando el aire acondicionado');
       return true;
  }
}
```

Ejemplo 24.1

```
class Engine implements Turnable {
  // ... otros miembros

  turnOn(): boolean {
       // ... implementación de la función...
       console.log('Encendiendo el motor');
       return true;
  }

  turnOff(): boolean {
       // ... implementación de la función...
       console.log('Apagando el motor');
       return true;
  }
}
```

Ejemplo 24.2

Al igual que ocurre con las clases, en el que ***un objeto puede manipularse considerando*** su tipo de dato o como si fuera alguno de sus ancestros (upcasting), también es posible tratarlos según ***cualquiera de las interfaces que implemente***.

Suele ser deseable programar pensando en interfaces ya que esto permite no ajustarse a un objeto en particular, sino a entidades que se comportan de un determinado modo, ***permitiendo que -en caso de ser necesario- pueda sustituirse el objeto subyacente por otro que se comporte de la misma manera***, evitando realizar modificaciones importantes en el código.

A veces este término se reduce a considerar clases abstractas en vez de clases abstractas.

Polimorfismo

En un sentido práctico aplicado a la programación, ***se trata de la capacidad que poseen los objetos de reaccionar de diversas maneras ante un mismo llamado, lo que ocurre como resultado de implementar comportamientos distintos en funciones que respetan la misma firma***.

Esto surge de la mano de la ***sobreescritura***, la implementación de ***interfaces*** y la ***composición*** de objetos.

La sobreescritura permite el polimorfismo dentro de una misma jerarquía, mientras que la implementación de una ***interfaz (o en su defecto extender una clase) permite involucrar objetos de distinta naturaleza***.

La ***composición***, en cambio, ***habilita que un objeto adopte distinto comportamiento en tiempo de ejecución***, según la implementación del objeto que lo compone, sin considerar la naturaleza de la jerarquía a la que pertenece el compuesto.

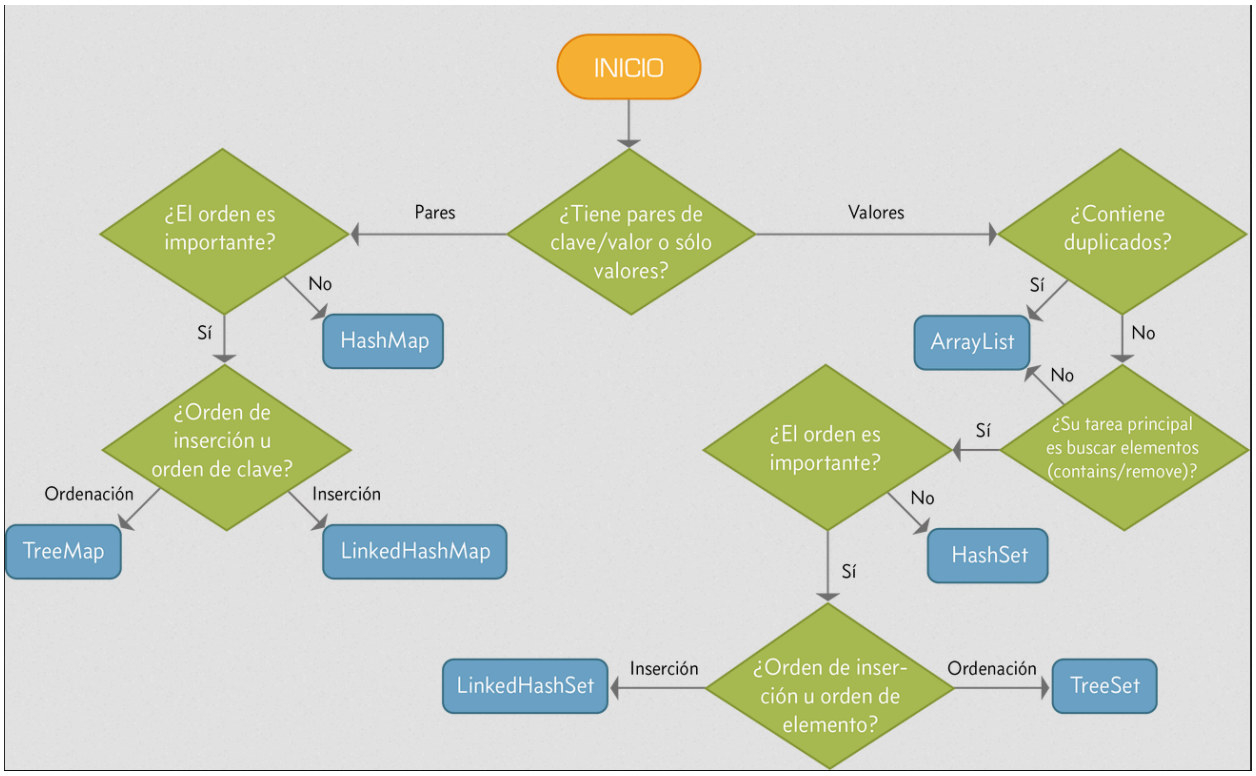
Colecciones

Se tratan de objetos que tienen la capacidad de agrupar a otros, todos ellos del mismo tipo y llamados genéricamente ***elementos***, a través de un conjunto de operaciones que

permiten la incorporación; adicionalmente, las colecciones incluyen funciones para manipular cada uno de los elementos.

A diferencia de un arreglo, que es otro objeto de tamaño fijo que permite el almacenado secuencial en memoria mediante mecanismos útiles pero rústicos, las listas y las colecciones poseen la facultad de autodimensionarse y ofrecer una mejor experiencia aunque, claro, sacrificando un poco más de recursos.

- **Set**: los objetos que implementan esta interfaz se caracterizan por no admitir duplicados.
    - HashSet:
    - LinkedHashSet:
    - TreeSet
- **List**: es uno de los más usados ya que permite el almacenamiento secuencial; se basa en un arreglo autodimensionable.
    - ArrayList:
    - LinkedList:
- **Map**: en otros lenguajes es conocido como diccionario, y se caracteriza por almacenar los valores considerando una llave por la cual se realizará la búsqueda; no admite duplicados.
    - HashMap:
    - TreeMap:
    - LinkedHashMap:



Fuente: https://www.adictosaltrabajo.com/2015/09/25/introduccion-a-colecciones-en-java/

Typescript

En typescript las colecciones de las que disponemos son:

- Array
- Map
- Set

Para utilizar cada una de estas colecciones, debemos indicar el tipo de dato que almacenará.

Array

Es el tipo de colección más simple y el más utilizado (probablemente). Este tipo de colección nos permite definir una lista de elementos de tamaño variable.

```
// Array de numeros
const numeros: Array<number> = new Array();

// Agrega un elemento al final
numeros.push(1);
numeros.push(10);

// Remueve un elemento del final
numeros.pop();

// Recorrer un Array
for(const numero of numeros) {
   console.log(numero);
}
```

Ejemplo 25.1

Para más información sobre las operaciones que se pueden realizar sobre un Array consultar: Array

Map

Esta estructura de datos permite almacenar elementos de tipo clave - valor. La característica distintiva es que una clave sólo puede aparecer una vez. Podemos pensar esta estructura como un diccionario.

Cabe destacar que tanto las claves como los valores pueden ser de cualquier tipo, inclusive objetos personalizados.

```
// Crear un mapa cuyas claves son de tipo string y sus valores de tipo number
const agenda: Map<string, number> = new Map();

// Agrega un elemento
agenda.set("Juan", 1234);
agenda.set("Ana", 5678)

// Obtener un elemento de un map
agenda.get("Ana")

// Saber si un elemento existe en el map
agenda.has("Ana")

// Eliminar un elemento del map
agenda.delete("Juan")

// Recorrer un Map
agenda.forEach((value, key) => {
  console.log(`Nombre: ${key}, Número: ${value}`);
```

```
});
```

Ejemplo 25.2

Para más información sobre las operaciones que se pueden realizar sobre un Map consultar: Map

Set

Como mencionamos anteriormente, un *Set* es un tipo de estructura de datos que se caracteriza por no admitir duplicados.

```
// Crear un set
const unicos: Set<number> = new Set();

// Agrega un elemento
unicos.add(1234);
unicos.add(5678);

// Saber si un elemento existe en el set
unicos.has(1234)

// Eliminar un elemento del Set
unicos.delete(1234)

// Recorrer un Set
agenda.forEach(unico => {
  console.log(unico);
});
```

Ejemplo 25.3

Por defecto, los Sets en TypeScript, mantienen la unicidad de los elementos que almacena, basándose en la referencia del objeto y no en sus valores, por lo que para determinar si dos objetos son iguales utiliza las referencias a memoria. Esta igualdad funciona bien para tipos de datos nativos como: *string*, *boolean* y *number* pero no funciona con tipos personalizados como pueden ser los objetos propios de la solución.

Para lograr que un *Set* que almacena objetos propios posea dicha característica podemos utilizar el siguiente código:

```
class GenericSet <T> {
```

```
  private items: T[];
  private getKey: (x:T) => string;

  constructor(comp: (x:T) => string);
  constructor(comp: (x:T) => string, values: T[]);
  constructor(comp: (x:T) => string, values?: T[]) {
    this.getKey = comp;
    this.items = [];
    if(values) {
      for(const v of values){
        this.add(v);
      }
    }
  }

  public has(item: T): boolean {
    return this.items.some(x => this.getKey(x) === this.getKey(item))
  }

  public get(key: string): T {
    return this.items.find(x => this.getKey(x) === key) as T;
  }

  public add(item: T): void {
    if(!this.has(item)) {
       this.items.push(item);
    }
  }

  public delete(item: T) {
    if(this.has(item)) {
       const aux = this.items.filter(x => this.getKey(x) !== this.getKey(item));
       this.items = aux;
    }
  }

  public values(): T[] {
    return [...this.items];
  }


  public forEach( cb: (element: T, index: number, array: T[]) => void) : void {
    for (let i = 0; i < this.items.length; i++){
       cb(this.items[i], i, this.items);
    }
  }
}
```

Ejemplo 25.4

Veamos un ejemplo de su uso

```
class Persona {
  constructor(private id: number, private name: string) {};

  public sayHi(): string {
       return `Hola ${this.name}`;
  }

  public getKey(): string {
       return `${this.id}`;
  }

}

const unicos = new GenericSet<Persona>(p => p.getKey());

unicos.add(new A(1, "Juan"));
unicos.add(new A(1, "Juan"));

for(const x of unicos.values()) {
  console.log(x)
}

console.log("----------------------------------------");

// Utilizando el constructor alternativo
const p3 = new Persona(3, "Juan");
const arrayUnicos = [new Persona(1, "Pedro"), new Persona(1, "Ana"), new Persona(2,
"Fede"), p3, new Persona(1, "Juli"), new Persona(4, "juan")];
const unicos = new GenericSet<Persona>(a => a.getKey(), arrayUnicos);
unicos.forEach((x) => console.log(x));

unicos.delete(p3);
unicos.forEach((x) => console.log(x));

console.log("----------------------------------------");

console.log(`Un elemento: ${JSON.stringify(unicos.get("2"))}`);
```

Ejemplo 25.5

El uso de la palabra Static

Para poder entender las implicancias de esta palabra reservada, es importante tener en cuenta que ***todo miembro que requiera de la existencia de un objeto para ser***

accedido, se lo conoce como miembro de instancia, mientras que aquel que no lo necesite, se lo conoce como miembro de clase.

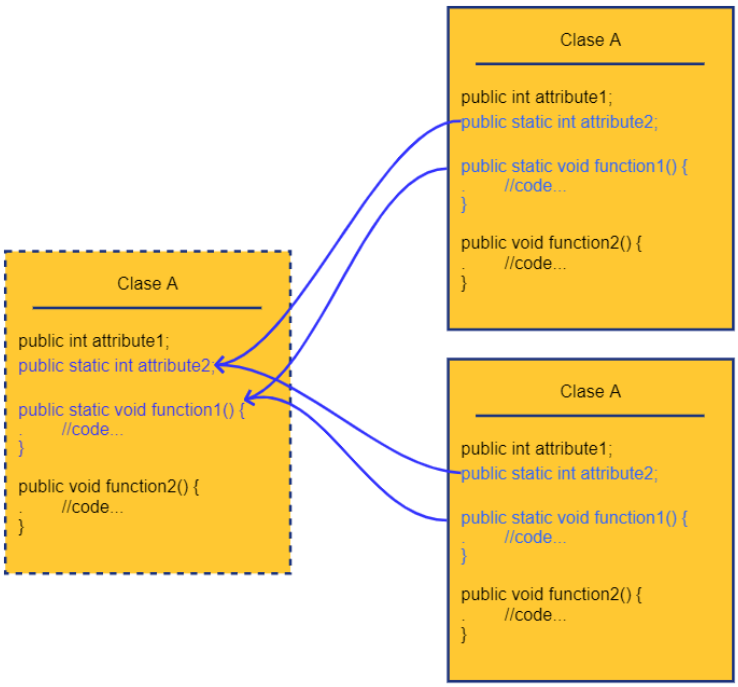
La palabra ***static*** es la directiva que ***se emplea para designar a los miembros de clase***, lo que implica que ***pueden ser invocados sin necesidad de que exista previamente una instancia***.

Retomando conceptos fundamentales, considerar que cada vez que se ejecuta un programa, se reserva suficiente espacio en memoria como para hacer una copia de todas las clases que se van a utilizar durante el ciclo de vida del programa, creando un repositorio virtual.

Luego, por cada nueva instancia que se necesite crear, el sistema operativo buscará en dicho repositorio la clase de la que se requiere una copia y procederá a su clonación, es decir, se reservará un nuevo espacio en memoria suficiente como para alocar la instancia.

Mientras todos los miembros no estáticos serán clonados al nuevo espacio, de aquellos marcados como estáticos únicamente se copiarán las referencias al espacio original, es decir, a la clase del repositorio virtual.

El siguiente gráfico ejemplifica conceptualmente lo que ocurre cuando se realizan instancias de una clase que presenta miembros estáticos; tanto attribute2 como function1 estarán apuntando al espacio de memoria original, mientras que el resto de los miembros serán alocados en su propio espacio.



Implicancia de los miembros estáticos

Como ocurre con todas las variables que apuntan a un mismo sector de memoria, cualquier cambio que suceda en éste será visible por todos los miembros que lo referencien debido a la centralización.

En el caso de las funciones este efecto no se advierte porque las instancias de un mismo tipo de clase siempre ejecutarán el mismo comportamiento, y cualquier subclase que se extienda y sobreescriba una función estática estará teniendo precedencia sobre el método original.

Es de vital importancia entender que ***todo miembro no estático puede acceder a un miembro estático pero no al revés***, ya que se presenta una suerte de relación uno a muchos; luego, los miembros estáticos pueden interactuar entre sí.

El siguiente ejemplo muestra cómo declarar una función estática.

```
class MyClass {
  public static performHandshake(): boolean {
       // ... implementación de la función...
       console.log('Realizando handshake...');
```

```
      // ... lógica para realizar el handshake ...
      return true; // O false si el handshake falla
  }
}
```

Ejemplo 31.1

A continuación se aprecia la declaración de dos atributos estáticos; estos aún pueden ser alterados ya que la palabra reservada static sólamente indica si pertenece al objeto o a la clase.

```
class Modem {
  public static BAUD_RATE_9600: number = 9600;
  public static BAUD_RATE_115200: number = 115200;

  // ... otros miembros
}
```

Ejemplo 31.2

Declaración de constantes

Comúnmente los atributos estáticos se emplean para definir constantes; combinar las palabras readonly y static en la declaración de una variable, resultará en la creación de un atributo que no puede ser alterado y que existe en un sólo lugar en memoria.

Un ligero cambio al ejemplo 31.2 permitirá que existan los atributos a nivel clase y que sea imposible modificarla, características que suelen ser convenientes para una constante.

```
class Modem {
  public static BAUD_RATE_9600: number = 9600;
  public static BAUD_RATE_115200: number = 115200;

  // ... otros miembros
}
```

Ejemplo 31.3

Enumerados

Son objetos que agrupan literales con el propósito de ser empleados como constantes y resultan ser una opción más formal que los atributos estáticos y finales; como es de esperarse, ***las clases del tipo enum no se pueden extender ni modificar dado que se emplean para tipar aspectos dentro de un rango de valores fijos***.

Por convención, toda constante debe ser escrita en mayúscula, separando las palabras que componen su literal mediante el uso de guiones bajo; en el ejemplo 31.3, en el que se declaran constantes utilizando atributos estáticos y finales, se aprecia claramente.

```
enum DayOfWeek {
  MONDAY,
  TUESDAY,
  WEDNESDAY,
  THURSDAY,
  FRIDAY,
  SATURDAY,
  SUNDAY,
}
```

Ejemplo 32.1

Su uso es muy sencillo.

```
import { DayOfWeek } from './entity/enums/DayOfWeek';

function main(): void {
  const selectedDay: DayOfWeek = DayOfWeek.MONDAY;

  switch (selectedDay) {
            case DayOfWeek.MONDAY:
              console.log("Es lunes");
              break;
            case DayOfWeek.TUESDAY:
              console.log("Es martes");
              break;
            // ... rest of the cases
            default:
               console.log("Día desconocido"); // Handle unexpected days
  }
}

main(); // Call the main function
```

Ejemplo 32.2

Errores y excepciones

En el ámbito del desarrollo, existen tres tipos de causas por las que una pieza de software puede fallar:

- **Errores sintácticos y semánticos**
- ***Errores de lógica***
- ***Eventos externos***

Los primeros ocurren en tiempo de diseño y son muy fáciles de detectar porque la mayoría de los IDE analizan el cumplimiento de las convenciones y estructuras en tiempo real; en el peor de los escenarios, se haría evidente al momento de compilar el código.

Los errores de lógica, en cambio, son mucho más difíciles de detectar porque cumplen con la sintaxis y semántica, y ***únicamente pueden ser descubiertos cuando el programa está en plena ejecución***.

Sin duda este es ***el caso más delicado*** porque dependiendo del comportamiento, ***podrían generarse resultados erráticos o inesperados con diversos grados de severidad***; los bugs, como se los suele llamar, habitualmente son clasificados como triviales, importantes, muy importantes y severos.

Algunos bugs podrían pasar desapercibidos durante mucho tiempo y sin impedir el funcionamiento de la solución informática; en cambio, otros, quizá causen que el sistema deje de funcionar tras generar resultados (datos o eventos) imposibles de tratar por un proceso posterior.

Una excepción ocurre cuando el flujo normal es alterado a raíz de un evento no contemplado, pudiendo deberse a un error de lógica u otro factor externo, resultando en la terminación inmediata del programa sin el adecuado tratamiento.

La jerarquía que Javascript define para el manejo de errores provee la clase base **Error.** A partir de la misma podemos derivar clases propias para realizar un manejo de errores más efectivo.

Control de excepciones

Para lidiar con las excepciones e impedir que el sistema quede en un estado inconsistente, la mayoría de los lenguajes de alto nivel implementan ***el bloque de control*** conocido como ***try/catch, que tiene por objetivo ejecutar un camino alternativo*** en caso de que un suceso poco habitual ocurra por alguno de los motivos mencionados anteriormente.

Por ejemplo, una división entre dos números podría desatar una excepción si el divisor adopta como valor el cero, debido a que no es posible efectuar tal operación aritmética; sin el adecuado control, el sistema terminaría inmediatamente.

La estructura de un bloque try/catch se compone de tres subbloques, uno de ellos opcional pero siempre recomendable:

- ***try***
  Este subbloque debe encerrar el algoritmo que se espera ejecutar en condiciones normales.

- ***catch***
  Debe contener el camino alternativo en caso de una excepción. Tener en cuenta que en el caso de los lenguajes tipados, suele ser obligatorio definir el tipo de excepción que se desea atrapar. Por lo general suelen ser adoptadas medidas correctivas y/o que anulan los resultados de los procesos anteriores; es habitual emplear este espacio para dejar registro del error.

- ***finally***
  Se trata de un subbloque opcional que, de estar presente, se ejecuta siempre independientemente de que se haya producido o no una excepción; un ejemplo podría ser el cierre de una conexión a la base de datos.

Es importante tener en cuenta que ***si un bloque try/catch intercepta una excepción, el código que sobrevenga al bloque catch, no se ejecutará***, de modo tal que toda acción correctiva eventual debe estar contemplada dentro del catch o del finally.

También es importante saber que ***siempre deben declararse primero los catch para las excepciones más específicas, dejando para lo último la más genérica***.

El siguiente fragmento de código muestra cómo el método ***getName()*** de la clase ***PersonManager*** extrae el nombre de la instancia ***Employee*** que recibe por parámetro, tomando la precaución de controlar a través del ***uso del bloque try/catch*** cualquier excepción.

Este caso es trivial porque el único modo en el que puede fallar es si el parámetro es null, situación que es posible controlar mediante la comparación contra la nulidad.

El subbloque ***catch actuará siempre y cuando la excepción ocurrida coincida con la declarada***; en este ejemplo se declaró la más genéricas de todas, razón por la que siempre actuará sin importar el evento que ocurra.

```
import { Employee } from '../entity/Employee';

class PersonManager {
  public getName(employee: Employee): string {
    try {
       const name = employee.getName();
       return name;
    }
    catch (error) {
       console.error('Error getting employee name:', error.message);
       return 'N/A'; // Default value if error occurs
    }
    finally {
       console.log('The finally block always executes.');
    }
  }
}
```

Ejemplo 35.1

Considerar el siguiente algoritmo.

```
import { PersonManager } from './manager/PersonManager';

function main(): void {
  const personManager = new PersonManager();
  const name = personManager.getName(null); // Might be null depending
  console.log(`Nombre: ${name}`);
}

main();
```

Ejemplo 37.1

La ejecución del código tendrá como resultado los siguientes renglones, siendo los primeros dos obra del bloque *try/catch* del método *getName()*.

```
Error getting employee name: <descripción del error>
The finally block always executes.
Nombre: N/A
```

Ejemplo 37.2

El parámetro del subbloque catch permite acceder a un conjunto de atributos y funciones que brindan datos útiles; una de ellas es el ***stack trace*** que imprime el registro a partir del punto en el que se produjo la excepción, elemento vital para identificar el problema.

Es habitual que el stacktrace se guarde en logs para su posterior análisis.

Lanzamiento de excepciones

Además de poder atrapar excepciones para evitar que el sistema termine abruptamente, es posible lanzar excepciones, de modo tal que los objetos que han invocado a una función que no puede continuar con su ejecución, tomen medidas al respecto.

Habitualmente las excepciones se lanzan si la pieza de software forma parte de una librería o si trabaja a muy bajo nivel, prácticamente en contacto con el hardware; en el primero de los casos es el modo más sencillo de pasar la responsabilidad hacia un nivel superior y, en el segundo, porque por lo general los sucesos no contemplados no siempre permiten una corrección o cambiar el curso de acción.

El lanzamiento de excepciones requiere incluir en la firma de la función desde la que se lanzará, la palabra reservada throw; en adición deben incluirse los tipos que podrían ser lanzados.

Adicionalmente, utilizar ***throw new*** en el código afectado para lanzar la interrupción.

```
import { Employee } from '../entity/Employee';

class PersonManager {
  public getLastName(employee: Employee | null): string {
    if (!employee) {
```

```
            throw new Error('Employee is null or undefined');
    }
    return employee.getLastName();
  }
}
```

Ejemplo 39.1

El ejemplo 39.1 muestra el lanzamiento de una excepción genérica en caso de que el el objeto employee sea nulo; el criterio que guíe el uso de esta herramienta está atada al diseño de la arquitectura del programa y de la estrategia que se utilice para el manejo de errores.

Excepciones personalizadas

Como mencionamos anteriormente, ***es posible crear tipos personalizados de excepciones*** que se ajusten a las necesidades de cada sistema ***con sólo extender la clase Error***.

Considerar las siguientes dos excepciones: ***EmployeeException*** y ***EmployeeAttributeException***.

```
export class EmployeeException extends Error {
  constructor(message: string) {
            super(message);
  }
}
```

Ejemplo 39.2

```
export class EmployeeAttributeException extends Error {
  constructor(message: string) {
          super(message);
  }
}
```

Ejemplo 40.1

Los dos ejemplos anteriores no hacen más que extender de la clase de excepción base, llamada ***Error*** e implementar uno de los constructores; se recomienda buscar en la documentación las especificaciones de la clase ***Error***.

Como toda extensión, es posible aggiornar las excepciones personalizadas con más atributos, parámetros y todo aquello que se considere necesario.

A continuación se muestra el ejemplo 39.1 modificado para adecuarlo al uso de dos excepciones; notar que la firma del método identifica cada una de las excepciones que lanza para que puedan ser identificadas adecuadamente.

El criterio para el uso de estas dos excepciones fue arbitrario con el fin de ilustrar un ejemplo.

```
import { Employee } from '../entity/Employee';
import { EmployeeAttributeException, EmployeeException } from '../exception';

class PersonManager {
  public getLastName(employee: Employee | null): string {
    if (!employee) {
       throw new EmployeeException('Employee is null or undefined');
    }

    if (!employee.getLastName().trim()) {
       throw new EmployeeAttributeException('Employee last name is empty or
contains only whitespace.');
    }

    return employee.getLastName();
  }
}
```

Ejemplo 40.2

A raíz de los cambios realizados a la clase ***PersonManager***, la función que invoque al método ***getLastname()*** tendrá que optar por atrapar la excepción (A) o burbujear (B) a un nivel superior las excepciones.

```
import { PersonManager } from './manager/PersonManager'; // Assuming PersonManager
is in manager folder
import { EmployeeException, EmployeeAttributeException } from './exception'; //
Assuming exceptions are in exception folder

function main(): void {
  try {
          const personManager = new PersonManager();
                const lastName = personManager.getLastName(null); // Might throw
exceptions
          console.log(`Nombre: ${lastName}`);
  }
  catch (error) {
          // Handle both exception types in a single catch block
          if (error instanceof EmployeeException) {
            console.error('Employee related error:', error.message);
          }
          else if (error instanceof EmployeeAttributeException) {
            console.error('Employee attribute error:', error.message);
          }
          else {
            console.error('Unexpected error:', error);
          }
  }
}

main();
```

Caso A - Ejemplo 41.1

```
import { PersonManager } from './manager/PersonManager'; // Assuming PersonManager
is in manager folder
import { EmployeeException, EmployeeAttributeException } from './exception'; //
Assuming exceptions are in exception folder

function main(): void {
  try {
       const personManager = new PersonManager();
       const lastName = personManager.getLastName(null); // Might throw exceptions
       console.log(`Nombre: ${lastName}`);
  }
  catch (error) {
       // Handle both exception types in a single catch block
       if (error instanceof EmployeeException) {
         console.error('Employee related error:', error.message);
```

```
      } else if (error instanceof EmployeeAttributeException) {
        console.error('Employee attribute error:', error.message);
      } else {
        console.error('Unexpected error:', error);
      }
  }
}

main();
```

Caso B - Ejemplo 41.2

Algo muy importante a tener en cuenta es que ***de lanzarse excepciones dentro de un bloque try*** resultaría en el atrapamiento inmediato por el subbloque ***catch*** de la misma función.

Unidad Temática N°2

~ Principios S.O.L.I.D. ~

"Cualquiera puede escribir código que una computadora pueda entender. Los buenos programadores escriben código que los humanos pueden entender.".

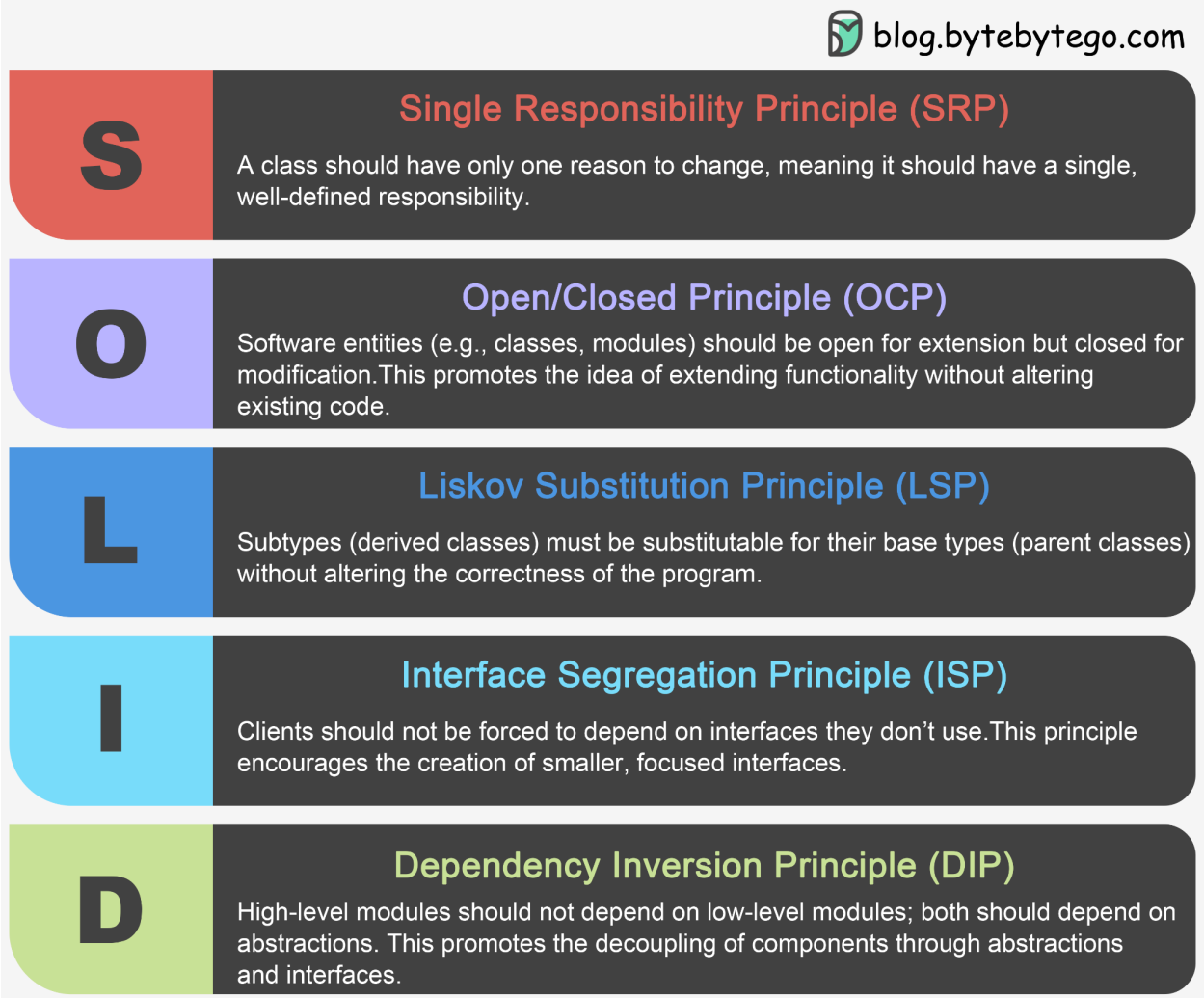
~ Martin Fowler

Principios S.O.L.I.D.

Introducción

Los principios S.O.L.I.D, son 5 postulados que fueron mencionados por primera vez al principio de los 2000 por Robert C. Martin en la presentación del paper Design Principles and Design Patterns.

Luego estos conceptos fueron desarrollados por Michael Feathers, dándole el nombre de principios SOLID.


blog.bytebytego.com
S
Single Responsibility Principle (SRP)
A class should have only one reason to change, meaning it should have a single, well-defined responsibility.
O
Open/Closed Principle (OCP)
Software entities (e.g., classes, modules) should be open for extension but closed for modification.This promotes the idea of extending functionality without altering existing code.
L
Liskov Substitution Principle (LSP)
Subtypes (derived classes) must be substitutable for their base types (parent classes) without altering the correctness of the program.
I
Interface Segregation Principle (ISP)
Clients should not be forced to depend on interfaces they don't use.This principle encourages the creation of smaller, focused interfaces.
D
Dependency Inversion Principle (DIP)
High-level modules should not depend on low-level modules; both should depend on abstractions. This promotes the decoupling of components through abstractions and interfaces.

Importancia de los principios SOLID en el desarrollo de software

Los principios de diseño, como los principios SOLID, desempeñan un papel fundamental en el proceso de desarrollo de software por varias razones:

Mantenibilidad: seguir unos principios de diseño sólidos hace que el código sea más fácil de mantener. Cuando el código está bien estructurado y se adhiere a estos principios, resulta más fácil identificar y solucionar problemas, añadir nuevas funciones y realizar mejoras sin causar consecuencias no deseadas.

Escalabilidad: Un software bien diseñado es escalable. Puede adaptarse a los cambios y al crecimiento de los requisitos sin necesidad de grandes modificaciones ni volverse cada vez más complejo.

Reusabilidad: La adhesión a los principios de diseño a menudo conduce a un código que es más reutilizable. Los componentes reutilizables ahorran tiempo y esfuerzo en el desarrollo y las pruebas.

Colaboración: Los principios de diseño proporcionan un marco común de trabajo para los desarrolladores. Este entendimiento común fomenta la colaboración y reduce los malentendidos entre los miembros del equipo.

Reducción de errores: Seguir los principios de diseño ayuda a identificar y mitigar los errores comunes de programación y los fallos de diseño. El resultado es un menor número de errores y un software más robusto.

Principios S.O.L.I.D

Single Responsibility (Responsabilidad única)

Este principio establece que una clase sólo debe tener una responsabilidad, esto nos lleva a pensar que por tener una única responsabilidad sólo debe tener una única razón para cambiar, para ser modificada.

Beneficios obtenidos al aplicar esta principio:

Pruebas: una clase con una única responsabilidad tendrá (probablemente) menos casos de prueba.

Reducción del acoplamiento: menos funcionalidad en una sola clase tendrá menos dependencias hacia otras clases.

Ejemplo

```
class Book {
  private name: string;
  private author: string;
  private text: string;

  // constructor, getters and setters

  public replaceWordInText(word: string): string {
       return this.text.replace(word, this.text);
  }

  public isWordInText(word: string): boolean {
       return this.text.includes(word);
  }

  public printTextToConsole(): void {
       // print the text to the console
  }
}
```

Ejemplo 42.1

¿Cuál es el problema en la clase Book?

Que no sólo manipula el contenido del libro sino que además permite formatear e imprimir su contenido.

¿Qué ocurre si se requiere cambiar el formato? ¿Qué pasa si se requiere imprimir a un archivo?, etc.

¿Cómo se soluciona?

```
class Book {
  private name: string;
  private author: string;
  private text: string;

  // constructor, getters and setters
```

```
  public replaceWordInText(word: string): string {
       return this.text.replace(word, this.text);
  }

  public isWordInText(word: string): boolean {
       return this.text.includes(word);
  }
}

class BookPrinter {
  // methods for outputting text
  public printTextToConsole(text: string): void {
       // print the text to the console
  }

  public printTextToAnotherMedium(text: string): void {
       // print the text to another device
  }
}
```

Ejemplo 42.2

Open for extension, Close to modification (abierto a la extensión, cerrado a la modificación)

Este principio menciona que las clases deben estar abiertas para su extensión pero cerradas para su modificación. Si hacemos esto, evitamos modificar el código existente y causar potenciales errores en una aplicación.

La excepción a la regla recae sobre la corrección de errores.

Ejemplo

Continuando con el ejemplo 42.1 y 42.2, suponga que se agrega el requerimiento de guardar en un archivo de texto el contenido del libro. Tendríamos el siguiente código:

```
class BookPersistence {
  // methods for outputting text
  public saveTextToFile(book: Book): void {
       // save the book's text to a text file
  }
}
```

Ejemplo 42.3

Tiempo después se solicita incorporar una base de datos y guardar el texto del libro en la misma para mejorar las búsquedas.

El primer acercamiento a la solución podría ser agregar un método en la clase *BookPersistence* que permita guardar el libro en una base de datos. Pero esto, claro está, rompería el principio Open/Close.

¿Cómo se soluciona?

El uso de interfaces (o en su defecto clases abstractas) podría ayudar a alcanzar la solución.

```
interface BookPersistence {
  save(book: Book): void;
}

class TextFilePersistence implements BookPersistence {
  public save(book: Book): void {
       // save the book's text to a text file
  }
}

class DbPersistence implements BookPersistence {
  public save(book: Book): void {
       // save the book's text to a database
  }
}
```

Ejemplo 42.4

En el ejemplo se puede apreciar que se tuvo que refactorizar el código ya que en el diseño inicial no se tuvo presente el principio Open/Close, pero el código que guarda el texto a un archivo de texto no fue modificado.

Otra ventaja es que ahora la aplicación puede ser extendida, agregando nuevas formas de guardar el contenido de un libro sin alterar el código existente.

Por último, no hay que perder de vista la ganancia obtenida de **desarrollar orientado a interfaces**. El código automáticamente pasó a ser polimórfico, con lo que podemos intercambiar las clases *TextFilePersistence* y *DbPersistence* y todas las que se agreguen sin que el código cliente se vea afectado.

Liskov Substitution (Sustitución de Liskov)

Bárbara Liskov introdujo este principio en 1987, y para explicarlo de forma sencilla, podemos decir que este principio establece que si B es una subclase de A, A debe poder ser reemplazada por B sin que la aplicación vea afectada su comportamiento.

El objetivo final de este principio, es poner de manifiesto el uso de clases polimórficas para clases que pertenezcan a una determinada jerarquía. Esto nos daría la ventaja de poder extender la aplicación sin que el código cliente se vea afectado.

```
abstract class Animal {
  public abstract makeNoise(): void;
}

class Dog extends Animal {
  public makeNoise(): void {
       console.log("bow wow");
  }
}

class Cat extends Animal {
  public makeNoise(): void {
       console.log("meow meow");
  }
}

class DumbDog extends Dog {
  public makeNoise(): void {
       throw new Error("I can't make noise");
  }
}
```

Ejemplo 42.5

En este ejemplo vemos que no se cumple el principio de Liskov, ya que si usaramos la una instancia de la clase `DumbDog` para reemplazar la clase `Dog`, nuestra aplicación dejaría de funcionar.

Interface Segregation (Separación de interfaces/conceptos)

Este principio simplemente significa que las interfaces más grandes (con muchos métodos) deben dividirse en otras más pequeñas. Al hacerlo, podemos asegurarnos de que las clases que las implementan sólo necesitan preocuparse por los métodos que les

interesan. En otras palabras, no forzar la implementación de métodos que una clase no necesita tener.

```
interface IBookPrinter {
  print(text: string): void;
}

interface IPaperBookPrinter {
  hasPaper(): boolean;
}

public class BookPrinter implements IBookPrinter{

  print(text: string): void{
       // imprime el código por consola
  }

}

public class PaperBookPrinter implements IPaperBookPrinter, IBookPrinter{

  hasPaper(): boolean{
       // return true o false si tiene o no papel
  }

  void print(String text){
       // imprime el código por consola
  }
}
```

Ejemplo 42.6

Dependency Inversion (Inversión de dependencia)

El último principio sugiere que las clases deben depender de abstracciones y no de clases concretas.

Este principio enfatiza la necesidad de tener interfaces que nos ayuden a definir y comunicarnos con las clases concretas.

La ventaja de esto es que se oculta la implementación real de la clase A a la clase B. Entonces, si la clase A cambia, la clase B no necesita ser modificada.

Una vez más lo que tratamos de hacer es desacoplar el código.

Veamos un ejemplo concreto de esto.

```
class Computer {
  private readonly keyboard: StandardKeyboard;
  private readonly monitor: Monitor;

  constructor() {
       this.monitor = new Monitor();
       this.keyboard = new StandardKeyboard();
  }
}
```

Ejemplo 42.7

Si bien este ejemplo funciona, podemos observar 2 problemas:

1. Estamos acoplando las clases Monitor, StandardKeyboard y Computer
2. Si necesitamos "actualizar" nuestro monitor o nuestro teclado, no podremos.

¿Cómo se soluciona?

```
interface Keyboard {}

interface Monitor {}

class Computer {
  private keyboard: Keyboard;
  private monitor: Monitor;

  constructor(keyboard: Keyboard, monitor: Monitor) {
       this.keyboard = keyboard;
       this.monitor = monitor;
  }
}
```

Ejemplo 42.8

Ahora las clases están desacopladas. Si queremos, podemos cambiar fácilmente el tipo de teclado en nuestra máquina con una implementación diferente de la interfaz. Podemos seguir el mismo principio para la clase Monitor.

Unidad Temática N°3

~ Estructuras de datos ~

"Si ha elegido las estructuras de datos correctas y las cosas están bien organizadas, los algoritmos casi siempre serán evidentes.".

~ Rob Pike

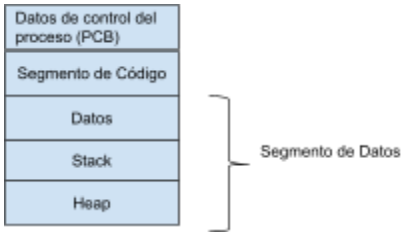
Estructuras de datos

En esta unidad vamos a estudiar el manejo de estructuras de datos dinámicas.

¿Por qué dinámicas? Porque a diferencia de los vectores y matrices cuyo tamaño es fijo y no varía durante la ejecución del programa, las estructuras dinámicas pueden crecer o reducirse a medida que se ejecuta el programa.

Antes de empezar a analizar cada una de las estructuras, vamos a aclarar algunos conceptos sobre la memoria.

Cuando un proceso es ejecutado, el mismo se divide en 2 grandes segmentos: código y datos. A su vez, el segmento de datos se subdivide en: datos variables, stack y heap.



- Código: contiene el código ejecutable del programa
- Datos: contiene variables locales globales y estáticas.
- Stack: utilizado para “apilar” las llamadas a funciones y sus parámetros.
- Heap: contiene variables cuya reserva de memoria es dinámica. Este segmento puede crecer o disminuir su tamaño.

El *stack* (o pila) es la porción de memoria que el sistema operativo utiliza para almacenar las llamadas a función y crear las variables locales de la función y sus parámetros. Es una porción de memoria que el sistema operativo maneja con un algoritmo tipo LIFO (último entrado, primero en salir) por lo que decimos que “apila” las llamadas a funciones. Cuando invocamos a una función, la misma (junto con sus parámetros y variables locales) se crean en el stack. Cuando la función retorna, todas las variables creadas en el stack son destruidas liberando el espacio utilizado.

Las estructuras dinámicas estarán alojadas en el segmento denominado *heap*. Este segmento es parte de la memoria RAM y pertenece al espacio de memoria asignado al proceso que se está ejecutando. A diferencia del *stack* no hay ningún algoritmo para almacenar los datos y la responsabilidad por la reserva y liberación de memoria se delega al programador.

La descripción anterior, refiere al comportamiento de una aplicación desarrollada utilizando lenguajes como C o C++ donde es posible utilizar funciones (brindadas por el lenguaje) que permiten manipular la memoria del proceso a bajo nivel.

En lenguajes de alto nivel como Java, Pyton, Javascript esto no es necesario ya que existen otros mecanismos automáticos que gestionan el uso de la memoria del proceso.

Las estructuras dinámicas que estudiaremos son: pilas, colas y listas.

Tipo de dato

Si bien, estas estructuras pueden almacenar cualquier tipo de dato, generalmente crearemos una clase a la que llamaremos *Nodo*.

La forma más genérica de definir esta clase es la siguiente:

```
class MyNode<T> {
  private _value: T;
  private _next: MyNode<T>;

  constructor(value: T) {
       this._value = value;
       this._next = undefined as unknown as MyNode<T>;
  }

  public get value(): T {
       return this._value;
  }

  public get next(): MyNode<T> {
       return this._next;
  }

  public set next(n: MyNode<T>) {
       this._next = n;
  }
}
```

Ejemplo 43.1

Nótese el uso del tipo de dato genérico (generics) que nos permite construir nodos que almacenan valores de tipo number, string u objetos de otro tipo.

Pilas

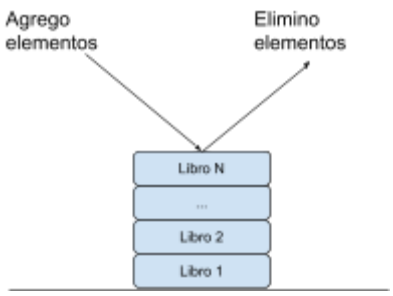
Son estructuras de datos que almacenan y recuperan sus elementos siguiendo un orden estricto. Este orden es *LIFO,* es decir, último en entrar primero en salir.

Definición	es una colección de elementos de un mismo tipo de dato a los que sólo se puede acceder (introducir y eliminar) por un único lugar denominado cabeza de pila.

En general, el uso de este tipo de estructura es adecuado cuando necesitamos invertir el orden de un conjunto de elementos o bien conocer el orden en que se fueron sucediendo determinados eventos, como por ejemplo el historial de navegación de páginas web, el historial de acciones de un programa para luego poder “deshacerlas”, etc.

Es importante destacar que la lectura sobre este tipo de estructura de datos es destructiva, es decir, para leer un elemento debemos removerlo de la pila.

Imaginemos una pila de platos o libros, el primer plato que agrego a la pila será el último que pueda extraer de la misma.



En este ejemplo, el primer libro agregado a la pila es el “Libro 1”, el segundo es el “Libro 2”, y así sucesivamente hasta el “Libro N”.

Cuando necesitemos leer los elementos de la pila, lo haremos en orden inverso, primero vamos a leer el “Libro N” y por último el “Libro 1”.

Definición de la clase pila

```
class Stack<T> {
 private head: MyNode<T>;

 public constructor() {
   this.head = undefined as unknown as MyNode<T>;
 }

 public push(value: T): void {
```

```
   const newNode = new MyNode(value);
   newNode.next = this.head;
   this.head = newNode;
 }

 public pop(): T | undefined {
  if (this.isEmpty()){
   return undefined;
  }

  const value = this.head.value;
  this.head = this.head.next;
  return value;
 }

 public isEmpty(): boolean {
   return this.head === undefined;
 }
}
```

Ejemplo 43.2

Operaciones sobre una pila

create (crear):

Para crear una pila no hay que realizar ninguna acción más que inicializar la cabecera de la pila, por lo que sólo debemos invocar al constructor de la clase.

isEmpty (está vacía)

Esta operación nos permite saber si la pila está vacía o no. Al igual que la operación de creación, es muy simple ya que sólo debemos verificar si la cabecera de la pila es *undefined*.

push (insertar)

La operación ***push*** nos permite insertar un elemento nuevo a la pila. Recordemos que la estructura de datos pila siempre inserta elementos en el tope de la pila, es decir en la cabecera.

pop (eliminar)

La operación ***pop*** nos permite eliminar un elemento de la pila. Recordemos que la estructura de datos pila siempre elimina elementos del tope de la pila, es decir de la cabecera.

Cabe aclarar que las operaciones de lectura sobre una pila son destructivas, en el sentido que para leer los elementos de la pila, debemos removerlos de la misma utilizando la función pop.

Tener en cuenta que antes de invocar a la función *pop* debemos verificar que la pila tenga elementos ya que de lo contrario producirá un error.

Ejemplo

```
// Crear una pila que almacena números
const p = new Stack<number>();

// Verificar estado de la pila. Salida esperada: true
console.log(p.isEmpty());

// Agregar elementos a la pila
p.push(1);
p.push(2);
p.push(4);

// Imprimir el contenido de la pila. Salida esperada 4, 2, 1
while(!p.isEmpty()) {
	console.log(p.pop());
}

// Verificar estado de la pila. Salida esperada: true
console.log(p.isEmpty());
```

Ejemplo 43.3

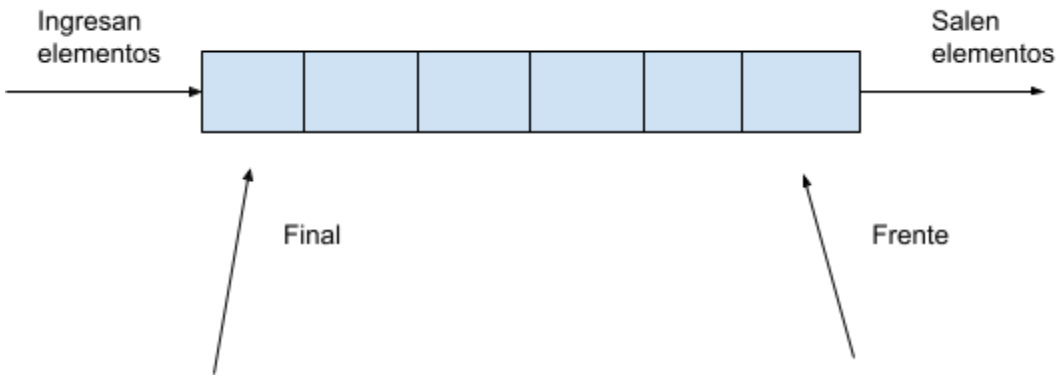
Colas

Al igual que las pilas, son estructuras de datos que almacenan y recuperan sus elementos siguiendo un orden estricto, pero a diferencia de éstas, el orden es *FIFO,* es decir, primero en entrar primero en salir.

En cierto modo, las *colas* son parecidas a las *pilas* en el sentido de que son una colección de elementos que se insertan por un extremo (parte final), pero la diferencia con estas últimas es que la extracción de dichos elementos no se realiza por el mismo extremo por el cual se insertaron sino que se realiza por el frente (parte inicial o cabecera).

Al igual que las pilas, la operación de lectura de los elementos es una operación destructiva ya que el elemento es removido de la cola.

Imaginemos una fila de personas que esperan para ser atendidas por el cajero de un banco o una lista de trabajos que esperan su turno para hacer uso del procesador.



Definición de la clase cola

```
class Queue<T> {
 private first: MyNode<T>;
 private last: MyNode<T>;

 public constructor() {
   this.first = undefined as unknown as MyNode<T>;
   this.last = undefined  as unknown as MyNode<T>;
 }

 public enqueue(value: T): void {
   const newNode = new MyNode(value);

   if (!this.first) {
       this.first = newNode;
   }
   else {
       this.last.next = newNode
   }

   this.last = newNode
 }

 public dequeue(): T | undefined {
  if (this.isEmpty()){
   return undefined;
  }
```

```
  const value = this.first.value;
  this.first = this.first.next;
  if (this.first === undefined) {
       this.last = undefined as unknown as MyNode<T>;
  }
  return value;
 }

 public isEmpty(): boolean {
   return this.first === undefined && this.last === undefined;
 }
}
```

Ejemplo 43.4

Operaciones sobre una cola

create (crear):

Para crear una pila no hay que realizar ninguna acción más que inicializar la los punteros de inicio y fin de la cola, por lo que sólo debemos invocar al constructor de la clase.

Para manejar una cola, utilizaremos una estructura que contenga ambos punteros, cabecera y final.

enqueue (agregar)

La operación *enqueue* nos permite agregar un elemento nuevo a la cola. Recordemos que la estructura de datos cola siempre inserta elementos en el final de la cola.

dequeue (eliminar)

La operación *dequeue* nos permite eliminar un elemento de la cola. Recordemos que la estructura de datos cola siempre elimina elementos del frente de la cola.

Cabe aclarar que las operaciones de lectura sobre una cola son destructivas, en el sentido que para leer los elementos de la cola, debemos removerlos de la misma utilizando la función remove.

Tener en cuenta que antes de invocar a la función *remove* debemos verificar que la cola tenga elementos ya que de lo contrario producirá un error.

isEmpty (está vacía)

Esta operación nos permite saber si la cola está vacía o no. Al igual que la operación de creación, es muy simple ya que sólo debemos verificar si los punteros de la cola son *undefined*.

Ejemplo

```
// Crear una cola que almacena números
const q = new Queue<number>();

// Verificar que esté vacía. Salida esperada: true
console.log(q.isEmpty());

// Encolar elementos
q.enqueue(1);
q.enqueue(2);
q.enqueue(4);

// Imprimir el contenido de la cola. Salida esperada 1, 2, 4
while(!q.isEmpty()) {
	console.log(q.dequeue());
}

// Verificar que esté vacía. Salida esperada: true
console.log(q.isEmpty());
```

Ejemplo 43.5

Listas

Una lista enlazada es una estructura de datos que maneja una colección de elementos dispuestos uno detrás de otro en la que cada elemento se conecta al siguiente mediante un puntero.

A diferencia de las *pilas* y *colas* no posee una política definida para el manejo de sus elementos, motivo por el cual una lista puede ser recorrida en cualquier orden, se puede insertar elementos en cualquier parte de la misma y las operaciones de lectura no son destructivas.

Al igual que con las otras estructuras de datos con las que estuvimos trabajando, para trabajar con listas debemos definir la estructura del nodo, que consta de datos y como mínimo un puntero al siguiente nodo.

Antes de comenzar a analizar las operaciones, hay que destacar que a diferencia de las estructuras vistas anteriormente, las operaciones que insertan un nodo en la lista también lo devuelven. Esta forma de trabajar nos va a permitir desarrollar estructuras más complejas como *listas de listas*.

Definición de la clase lista

```
class List<T> {
  private head: MyNode<T>;

  constructor() {
       this.head = undefined as unknown as MyNode<T>;
  }

  public push(value: T): MyNode<T> {
       const node = new MyNode(value);

       let headAux = this.head;
       while(headAux && headAux.next) {
       headAux = headAux.next;
       }
       if (!headAux) {
       this.head = node
       }
       else {
       headAux.next = node;
       }

       return node;
  }

  public pop(): T {
       let value = undefined;
       let headAux = this.head;
       let previous:MyNode<T> = undefined as unknown as MyNode<T>;
       while(headAux && headAux.next) {
       previous = headAux;
       headAux = headAux.next;
       }

       if (!previous) {
       this.head = undefined as unknown as MyNode<T>;
```

```
        }
        else {
        previous.next = undefined as unknown as MyNode<T>;
        }
        value = headAux.value;
        return value;
    }

    public insertFirst(value: T): MyNode<T> {
        const node = new MyNode(value);
        node.next = this.head;
        this.head = node;
        return node;
    }

    public removeFirst(): T {
        let value  = undefined as unknown as T;
        if (this.head) {
        value = this.head.value;
        this.head = this.head.next;
        }

        return value;
    }

    public insertOrdered(value: T): MyNode<T> {
        const node = new MyNode(value);
        let headAux = this.head;
        let previous:MyNode<T> = undefined as unknown as MyNode<T>;
        while(headAux && headAux.value < value) {
        previous = headAux;
        headAux = headAux.next;
        }

        if (!previous) {
        this.head = node;
        }
        else {
        previous.next = node;
        }
        node.next = headAux;

        return node;
    }

    insertUnique(value: T): MyNode<T> {
        const node = this.search(value);
        if (!node) {
              this.insertOrdered(value);
        }
        return node;
    }
```

```
  public delete(value: T): T {
       let headAux = this.head;
       let previous:MyNode<T> = undefined as unknown as MyNode<T>;
       while(headAux && headAux.value !== value) {
       previous = headAux;
       headAux = headAux.next;
       }

       if (!previous) {
       this.head = headAux.next;
       }
       else {
       previous.next = headAux.next;
       }

       return headAux.value;
  }

  public search(value: T): MyNode<T> {
       let headAux = this.head;
       while(headAux && headAux.value !== value) {
       headAux = headAux.next;
       }
       return headAux;
  }

  public sort(): void {
       let value: T;
       let listAux = new List<T>();
       while(this.head) {
              value = this.removeFirst();
              listAux.insertOrdered(value);
       }
       this.head = listAux.head;
       listAux.clear();
  }

  public clear(): void {
       while(this.head){
              this.removeFirst();
       }
  }
}
```

Ejemplo 43.6

Operaciones sobre una lista

Create (crear):

Para crear una lista no hay que realizar ninguna acción más que inicializar la cabecera de la lista, por lo que sólo debemos invocar al constructor de la clase.

insertFirst (inserta adelante)

Esta operación agrega un nodo delante del primer nodo.

push (inserta al final)

Esta operación agrega un nodo al final de la lista.

En esta función debemos destacar la utilización del puntero auxiliar *headAux*. Este puntero es utilizado para no perder la referencia al primer nodo de la lista.

insertOrdered (inserta ordenado)

Esta operación permite insertar un nodo conservando el orden, en otras palabras, esta función permite insertar un nodo respetando el orden de los datos.

search (buscar un nodo)

Esta función permite buscar y devolver el nodo que contenga el valor pasado por parámetro.

removeFirst (elimina el primer nodo)

Esta operación permite eliminar el primer nodo de la lista (similar al manejo de pilas)

pop (elimina el último nodo)

Esta operación permite eliminar el último nodo de la lista

delete (elimina un nodo particular)

Esta función permite eliminar el nodo que contiene el valor pasado por parámetro.

clear (vaciar la lista)

Esta operación permite vaciar la lista

SortList (ordenar una lista)

Esta función permite ordenar una lista. Para llevar a cabo esta tarea vamos a reutilizar las funciones *removeFirst* e *insertOrdered*.

insertUnique (inserta sin repetición)

Esta función permite insertar un nodo siempre y cuando el mismo no exista previamente en la lista. En caso de no existir lo inserta respetando el orden de la lista. Devuelve el puntero al nodo encontrado o insertado.

Unidad Temática N°4

~ Patrones de diseño ~

"Los patrones de diseño identifican los problemas de diseño recurrentes y sus soluciones."

~ Erich Gamma

Introducción a los patrones de diseño

A lo largo de la historia de la programación, los programadores han brindado solución a innumerables problemas, algunos de los cuales han sido -y siguen siendo- muy recurrentes.

Para evitar que los profesionales implementen estrategias que quizá no sean las mejores en términos de mantenibilidad, lectura y escalabilidad, el conocido grupo "Gang of Four" ha estado estudiando los problemas con mayor recurrencia y el modo en el que los desarrolladores supieron abordarlos, identificando los patrones de arquitectura que ofrecen mejores beneficios en relación con su implementación.

Tras un largo trabajo, en 1994 se edita el libro ***Design Patterns: Elements of Reusable Object-Oriented Software*** que resulta ser el compendio de las mejores prácticas en términos de diseño, identificando tres categorías de patrones:

- ***Creacionales***: estrategias para la creación de objetos.
- ***Comportamiento***: focaliza sobre la comunicación entre objetos.
- ***Estructurales***: aborda la relación entre los objetos.

Lo interesante de los patrones de diseño es que, en la mayoría de los casos, surgen como las propuestas más naturales para enfrentar un desafío, siempre y cuando el arquitecto se ciña a las buenas prácticas de la programación orientada a objetos.

Adicionalmente, el hecho de estar documentados permite que los responsables del desarrollo de una solución puedan manejar términos técnicos dentro de un mismo dominio, mejorando la comunicación.

Es importante tener presente que su mera existencia no implica su uso, más aún si las bases fundamentales de la programación orientada a objetos no han sido asimiladas; por otra parte, pensar que en los últimos veintiséis años el desarrollo de software ha madurado muchísimo y los patrones de diseños propuestos podrían no ser la respuesta más eficiente para los lenguajes modernos.

En lo particular, el libro habla sobre los errores comunes en la programación orientada a objetos, la importancia de programar pensando en interfaces y no en implementación, y el uso de la composición sobre la herencia.

Una lectura rápida sobre los temas que aborda el libro puede ser encontrada aquí: https://en.wikipedia.org/wiki/Design_Patterns

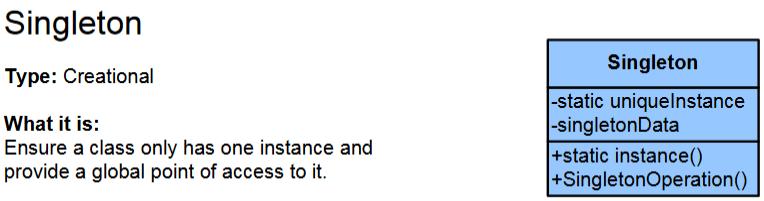
Patrones creacionales

Estos patrones son herramientas que facilitan la creación de las entidades, encapsulando el proceso de construcción; de este modo, si se necesitara cambiar la estructura de un objeto, sus requerimientos o la manera en que estos se componen, las modificaciones se verían muy acotadas.

Singleton

Se asegura que exista una única instancia de un determinado objeto, al controlar el modo en el que éste se crea; es más plástico que una clase del tipo final con funciones estáticas ya que puede ser extendida y sus métodos sobreescritos.

Uno de los posibles usos puede ser el mantener la configuración del sistema cargada en memoria y disponible para toda la pieza de software.



La clase afectada debe presentar el ***constructor con visibilidad privada*** para que no pueda ser creada accidentalmente desde afuera mediante la palabra reservada ***new***, una función estática que se encargue de instanciar el objeto y almacenarlo como atributo de sí mismo, y las funciones necesarias para brindar el servicio para el cual fue creado.

Resumen

Propósito	Asegura que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.
Aplicabilidad	• El motivo más común es controlar el acceso a algún recurso compartido. • Permite acceder y controlar el acceso a variables globales.

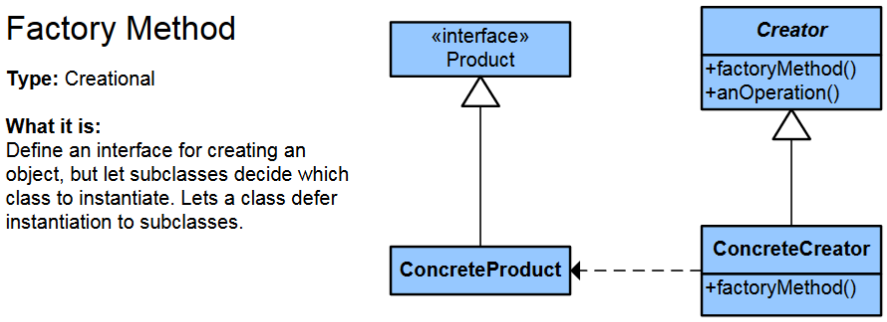
Ventajas	**Desventajas**
1. Se puede tener la seguridad de que la clase tiene una única instancia. 2. Se obtiene un único punto de acceso a dicha instancia.	1. Vulnera el Principio de responsabilidad única. 2. Singleton puede ocultar un mal diseño, vulnerando el Principio de Inversión de

3. El Singleton solo se inicializa realmente cuando se requiere la primera vez.	Dependencia. 3. En entornos con múltiples hilos de ejecución, requiere de un tratamiento especial para que varios hilos no creen un objeto Singleton varias veces.

Factory method

Es similar al Factory excepto que la formalidad de emplear interfaces permite que la creación de los objetos pueda tratarse de un modo más genérico y facilitar la automatización.

Los objetos creados obedecen a una interfaz (o clase abstracta) mientras que las "fábricas" se extienden de una clase abstracta que trabaja produciendo objetos que respetan dicha interfaz; son las subclases de la "fábrica" quienes deciden qué tipo de objeto construir.



Creator es la clase abstracta que define qué el método abstracto que deberá ser implementado por las subclases y que tendrá por objetivo devolver un objeto que responda a la interfaz ***Product***; dependerá de ***ConcreteCreator*** qué ***ConcreteProduct*** fabricar.

Resumen

Propósito	Proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán.
Aplicabilidad	Útil cuando no se conoce de antemano las dependencias y los tipos exactos de los objetos con los que debe funcionar tu código.

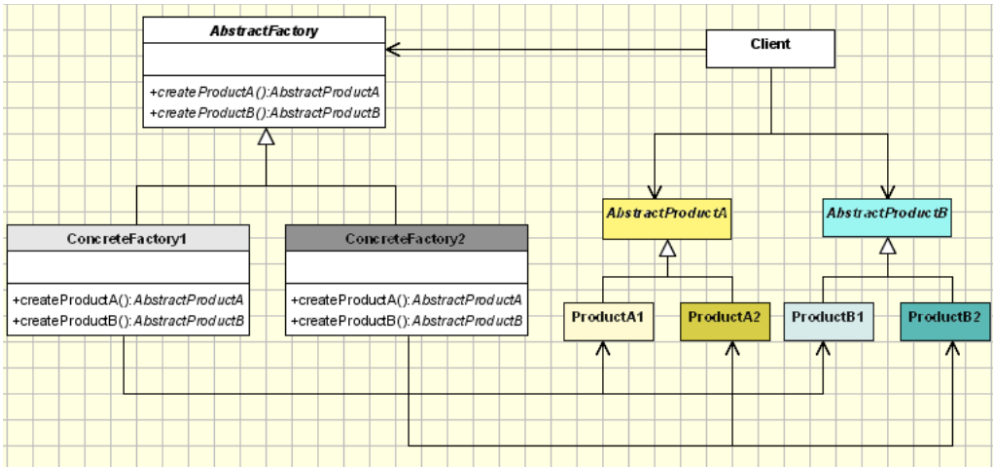
Ventajas	**Desventajas**
1. Evitas un acoplamiento fuerte entre el	1. Puede requerir la implementación de

creador y los productos concretos. 2. Principio de responsabilidad única. Puedes mover el código de creación de producto a un lugar del programa, haciendo que el código sea más fácil de mantener. 3. Principio de abierto/cerrado. Puedes incorporar nuevos tipos de productos en el programa sin descomponer el código cliente existente.	muchas subclases

Abstract factory

Mientras que el Factory Method ofrece la creación de un elemento por “fábrica”, el Abstract factory ofrece la construcción de los elementos que componen a una familia.

Por ejemplo, podría considerarse que la preparación de un cocktail contiene una base de alcohol y un adicional (jugo, gaseosa, etc); se podría emplear este patrón para preparar Gin & Tonic o Campari con jugo de naranja.



El diagrama muestra la interfaz (o clase abstracta) ***AbstractFactory*** que es la que define los lineamientos que deben ser implementados por ***ConcretFactory1*** y ***ConcretFactory2***; ambos crean componentes que respetan la interfaces (o clases abstractas) ***AbstractProductA*** y ***AbstractProductB***.

De este modo, cuando se cree la instancia de ***ConcretFactory1*** se podrán obtener ***ProductA1*** y ***ProductB1***, mientras que ***ConcretFactory2*** se podrán obtener ***ProductA2*** y ***ProductB2***.

Continuando con la analogía de los cocktails, ***AbstractProductA*** puede modelar las bebidas alcohólicas y ***AbstractProductB*** los agregados.

Resumen

Propósito	Permite crear familias de objetos relacionados sin necesidad de especificar sus clases concretas
Aplicabilidad	• Cuando el código deba funcionar con familias de productos relacionados, pero no desees que dependa de las clases concretas de esos productos, ya que puede ser que no los conozcas de antemano o sencillamente quieras permitir una futura extensibilidad.

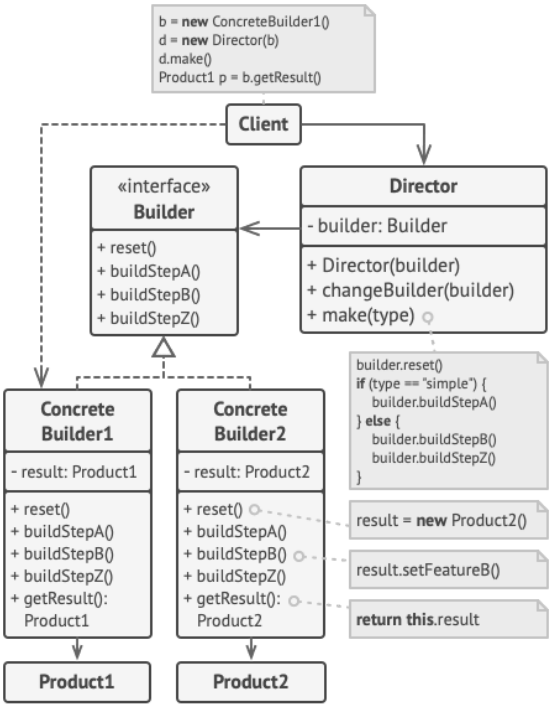
Ventajas	**Desventajas**
1. Los productos que se obtienen de una fábrica son compatibles entre sí. 2. Se evita un acoplamiento fuerte entre productos concretos y el código cliente. 3. Principio de responsabilidad única. Puedes mover el código de creación de producto a un lugar del programa, haciendo que el código sea más fácil de mantener. 4. Principio de abierto/cerrado. Puedes incorporar nuevos tipos de productos en el programa sin descomponer el código cliente existente.	1. Puede requerir la implementación de muchas clases, subclases e interfaces

Builder

Permite construir objetos complejos paso a paso. El patrón nos permite producir distintos tipos y representaciones de un objeto empleando el mismo código de construcción.

Suele ser útil cuando la construcción de un objeto depende de muchos parámetros y/o muchos pasos. Con este patrón podemos desacoplar la construcción de dicho objeto de su propia clase y se delegue en un objeto que llamaremos constructor (o builder).

Al organizar la construcción de un objeto en varios pasos, si no son todos obligatorios, podemos ejecutar unos y no otros con lo que obtendremos objetos con distintas configuraciones.



La interfaz Builder declara los pasos de construcción, que todos los objetos constructores deben implementar. Los Constructores Concretos ofrecerán diferentes implementaciones de los pasos de construcción. Los objetos resultantes, son los Productos.

En la clase Director, podemos implementar distintos métodos que nos permitan obtener objetos con distintas configuraciones. La clase Director define el orden en el que se invocarán los pasos de construcción, por lo que es posible crear y reutilizar configuraciones específicas de los productos.

El Cliente debe asociar uno de los objetos constructores con la clase director.

Resumen

Propósito	Construir objetos complejos paso a paso. El patrón nos permite producir distintos tipos y representaciones de un objeto empleando el mismo código de construcción
Aplicabilidad	• Evitar constructores complejos y con muchos parámetros. • Ser capaz de crear distintas representaciones de ciertos objetos.
Ventajas	**Desventajas**

1. Construir objetos paso a paso 2. Es posible reutilizar el mismo código de construcción para construir distintas representaciones de productos. 3. Principio de responsabilidad única: se puede separar un código de construcción complejo de la lógica de negocio del producto.	1. La complejidad general del código aumenta, ya que Builder implica la creación de varias clases nuevas.

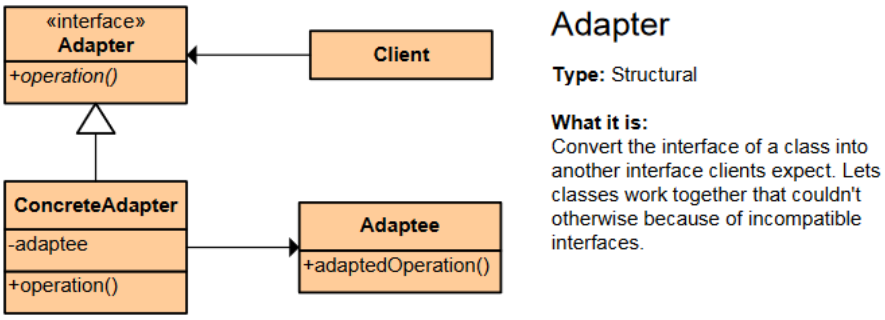
Patrones estructurales

Ofrecen estrategias para que los objetos pueden ser tratados uniformemente u organizarse de modo tal que resulte en una estructura práctica para la manipulación de las entidades.

Adapter

Encapsula el código de modo tal que un conjunto de objetos que exponen funciones diferentes (y que, por lo general, no pueden ser modificados) sean tratados mediante una interfaz común.

Suele ser habitual el uso de esta estrategia cuando se requiere explotar las funcionalidades de subsistemas o bibliotecas que realizan tareas similares, como los drivers de impresoras.

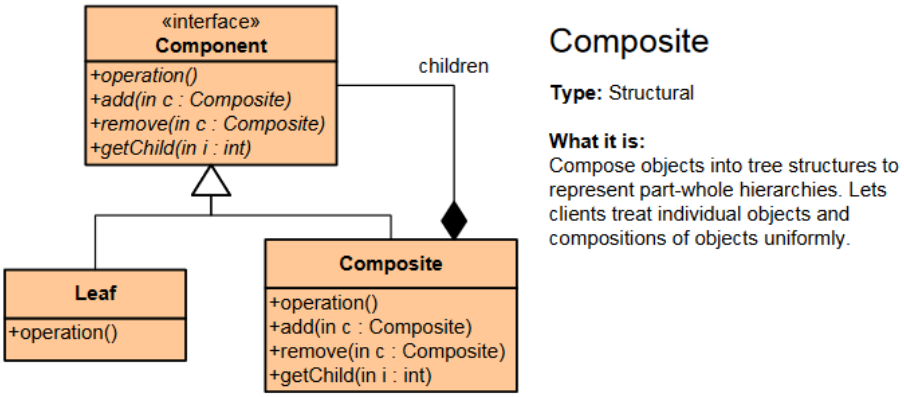


El diagrama muestra la interfaz llamada ***Adapter*** (podría ser una clase abstracta) con el método ***operation()*** que debe ser implementado por la clase adaptadora ***ConcreteAdapter***; ésta, a su vez está compuesta por la clase a adaptar llamada ***Adaptee*** que posee la función ***adapedOperation()***. ***ConcreteAdapter*** sólo necesita invocar a ***adapedOperation()*** desde ***operation()*** para hacer uso de la clase adaptada.

Composite

Se trata de un patrón muy utilizado para generar estructuras arborescentes o rizomáticas, como en el caso de los menúes que pueden contener opciones o un conjunto de ellas agrupadas en un submenú.

Su uso, puntualmente, encuentra su motivo cuando el comportamiento de las "ramas" y de las "hojas" son distintas; cuando esto no es así, sería suficiente implementar una clase con una lista de elementos de su mismo tipo.



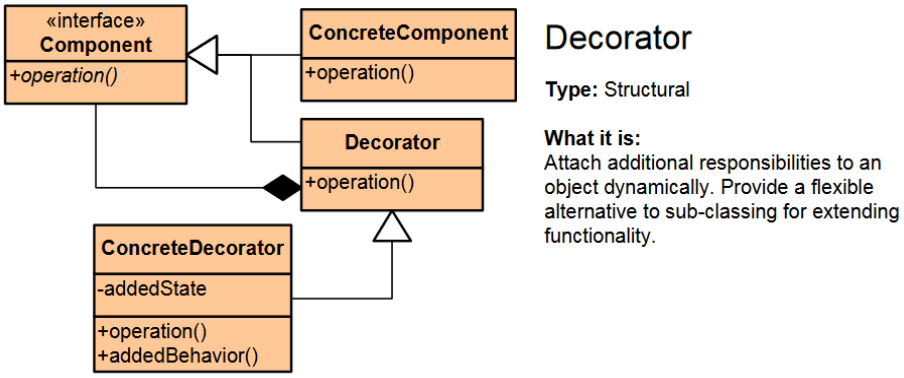
El diagrama exhibe la interfaz llamada ***Component*** (podría ser una clase abstracta) con varios métodos, algunos permiten trabajar sobre la estructura ***-add(), remove(), getChild()***- y otro propio de la tarea de componente, ***operation()***.

La clase ***Leaf*** u hoja, es una entidad que implementa únicamente el método ***operation()*** (todos aquellos definidos en la interfaz), mientras que la clase ***Composite*** es la rama que no sólo implementa ***operation()*** (todos aquellos definidos en la interfaz) sino que también posee una colección de objetos del tipo ***Component*** que ***pueden ser tanto ramas como hojas***, y que se manipulan mediante las funciones indicadas a priori.

Uno de los problemas que exhibe este patrón es que va en contra de la segregación de interfaces, ya que las clases hoja están obligadas a implementar las funciones para la manipulación de la estructura aún cuando no puedan agregar, eliminar u obtener elementos por ser terminal.

Decorator

El objetivo de este patrón es poder extender la funcionalidades de un objeto, evitando realizar cambios en estructuras complejas o imposibles de ser modificadas.



ConcreteComponent existe y ejecuta una determinada tarea; idealmente implementa la interfaz ***Component*** (podría ser una clase abstracta también).

Cuando llega el momento de agregar funcionalidades, se crea una clase abstracta ***Decorator*** que implementa (o se extiende, según el caso) de ***Component***, procurando incluir un atributo del mismo tipo para establecer una composición.

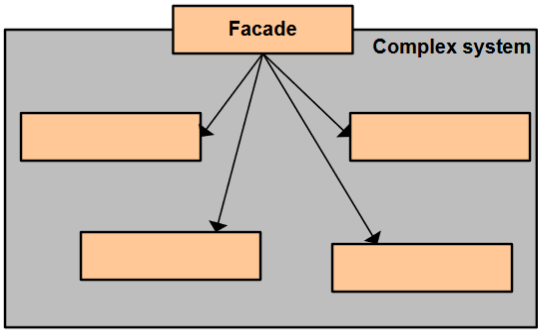
ConcreteDecorator extiende de ***Decorator*** agregando nuevas características, componiéndose por ***ConcretComponent***.

Finalmente, los llamados a ***ConcretComponent*** ahora se realizan a ***ConcretDecorator*** para que éste sea el que, oportunamente, invoque a ***ConcretComponent***; adicionalmente, en algún punto de la cadena de llamados, ***ConcretDecorator*** se encargará de ejecutar las nuevas tareas.

Facade

Conceptualmente hablando, es uno de los más sencillos, ya que propone esconder la complejidad de un sistema detrás de una clase o interfaz de alto nivel, encapsulando toda lógica.

La idea es poder ofrecerle al consumidor un conjunto acotado de funciones, cuya implementación sea el encargado de articular con los componentes a enmascarar.



Facade

Type: Structural

What it is:
Provide a unified interface to a set of interfaces in a subsystem. Defines a high-level interface that makes the subsystem easier to use.

Patrones de comportamiento

Los patrones de comportamiento ofrecen estrategias de comunicación entre los objetos para que las responsabilidades sean llevadas a cabo de forma práctica y eficiente para el mantenimiento de código.

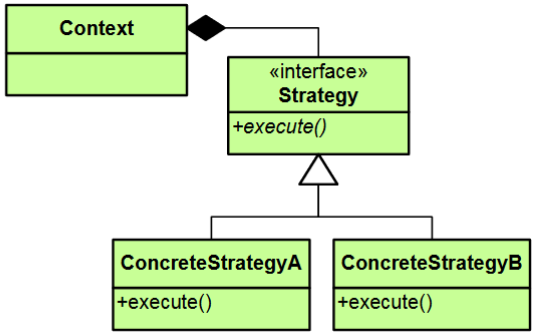
Strategy

Es uno de los diseños más simples y, a la vez, representativo del polimorfismo; propone la composición como herramienta para que los objetos cambien su comportamiento según los criterios que el programador defina.

Strategy

Type: Behavioral

What it is:
Define a family of algorithms, encapsulate each one, and make them interchangeable. Lets the algorithm vary independently from clients that use it.

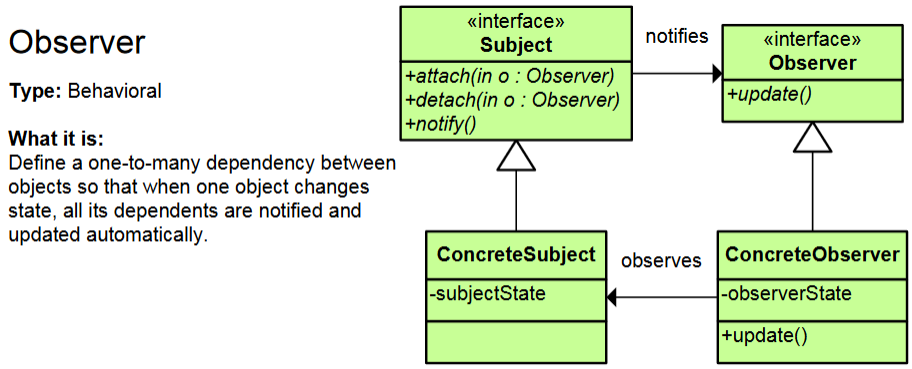


Context posee un atributo que se ciñe a la interfaz (o clase abstracta) ***Strategy***, responsable de definir el uso del método ***execute()*** y cuya implementación recae en las clases que la implementen, en este caso ***ConcreteStrategyA*** y ***ConcreteStrategyB***.
De este modo, es posible crear objetos similares que se comporten de distinta manera o, más poderoso aún, cambiar el comportamiento durante el tiempo de ejecución.

Observer

La arquitectura de este patrón permite que varios objetos puedan actuar en consecuencia cuando un tercero, el observado, adopta un determinado estado o realiza alguna acción específica.

A los efectos prácticos, el objeto observado es en realidad "el avisador", responsable de indicarles a los interesados que un determinado estado o evento ocurrió, para que ellos hagan lo que tengas que hacer.



ConcreteSubject es el objeto que será observado y cuyo cambio de estado es de interés para el observador ***ConcretObserver*** ya que, cuando el primero cambie, deberá realizar una acción.
Subject, la interfaz que implementa ***ConcreteSubject***, obliga a que se implemente la función ***attach()*** para la suscripción de los observadores, ***detach()*** para la desuscripción, y ***notify()*** para anoticiar a los observadores que el cambio de estado o evento ocurrió y, de este modo, que lleven a cabo las acciones pertinentes.
La interfaz ***Observer*** obliga a ***ConcretObserver*** a implementar el método ***update()*** que es el que será invocado por el observado cuando cambie de estado.
El funcionamiento es sencillo: ***ConcreteSubject*** guarda las referencias de todos los observadores mediante el método ***attach()*** y, cuando ocurra el cambio de estado, ejecutará el método ***notify()***, responsable de invocar las funciones update() de cada uno de los observadores suscriptos.
Debido a que ***update()*** no recibe parámetros y que el observador quizá necesite datos del observado, es habitual que el primero mantenga una referencia a través de la composición; otra alternativa es conseguir que ambos convivan en el mismo contexto para que el observador acceda a algún miembro del observado, pero esto último es poco frecuente.

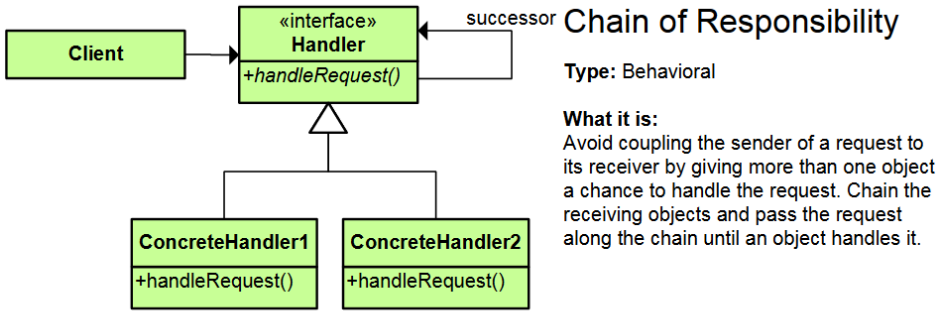
Chain of responsibility

Es similar al Decorator en cuanto a que depende del armado de una cadena de clases mediante el uso de la composición, cuyos métodos se van llamando unos a otros; no obstante la

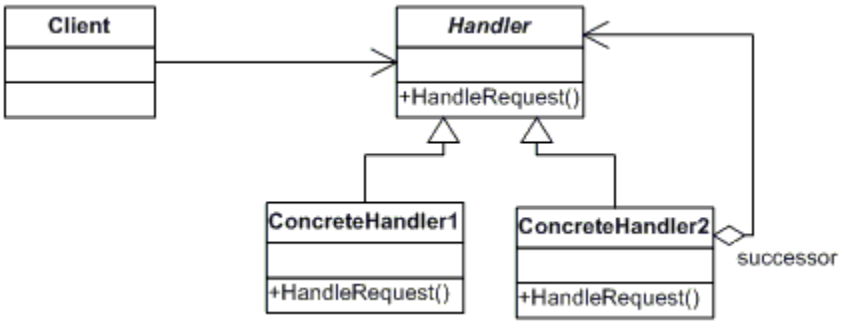
motivación es distinta: mientras Decorator se emplea para ir agregando funcionalidades nuevas a una clase que no puede o debe modificarse, Chain of responsibility es un “pasamanos” en el que las funciones se van pasando un mismo objeto y, aquella a la que le competa, realizará la acción que le corresponda.

Un ejemplo podría ser en una cadena de mando: la orden se va pasando de rango en rango hasta llegar al interesado, quien cumple con la orden.

El siguiente diagrama no es del todo preciso ya que la relación ***successor*** es, en realidad, una relación de agregación y de un elemento.


Client
«interface»
Handler
+handleRequest()
successor
ConcreteHandler1
+handleRequest()
ConcreteHandler2
+handleRequest()
Chain of Responsibility
Type: Behavioral
What it is:
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Otras implementaciones presentan la estructura similar a un Composite, buscando que el último responsable sea del tipo terminal; la agregación es de un sólo elemento, siempre.



Handler es la interfaz o clase abstracta que define la firma del método ***HandleRequest()*** y que, por lo general, recibe un objeto o ciertos parámetros a evaluar (también podrían ser pasados por constructor.

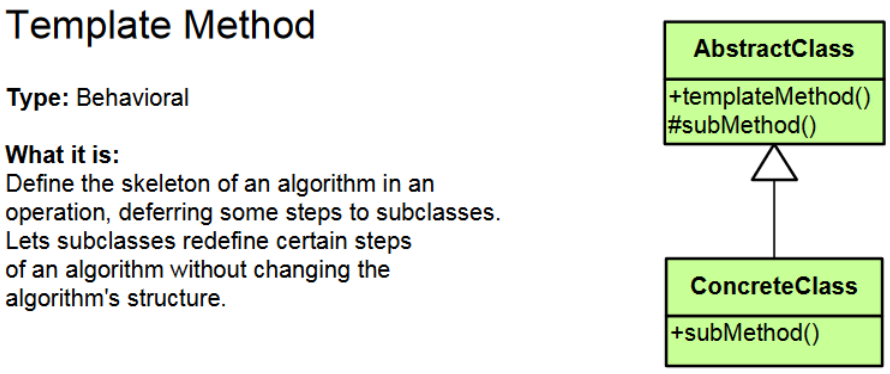
Concretehandler1 implementa o se extiende de ***Handler***, presentando una relación de agregación con otro objeto que también respeta la interfaz ***Handler***, por ejemplo ***Concretehandler2***, creando una cadena que puede extenderse tanto como sea necesario.

Cuando ***Client*** necesita realizar una operación, invoca a la función ***HandleRequest()*** de la clase ***Concretehandler1***, la que evaluará si tiene que realizar alguna tarea; luego, ésta invoca el método ***HandleRequest()*** de su clase agregada, acción que se repite hasta llegar al último eslabón.

Template Method

El patrón es útil cuando un proceso siempre requiere que se ejecuten los mismos pasos, pero admitiendo algunas variaciones en la implementación, todas ellas delegables a subclases.

Por ejemplo, una función que englobe el proceso de validación de datos de un objeto, el cumplimiento de las reglas de negocio, y la posterior persistencia en una base, no cambiará nunca su estructura; sin embargo, qué datos a validar, qué reglas de negocio aplicar y qué valores se persistirán, estarán en estrecha relación con la entidad a manejar. Este patrón permite es plasticidad.


Template Method
Type: Behavioral
What it is:
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
AbstractClass
+templateMethod()
#subMethod()
ConcreteClass
+subMethod()

AbstractClass es la clase abstracta que presenta la función ***templateMethod()*** con una determinada estructura, es decir, un conjunto de líneas de códigos que cumplen un propósito.
Una de esas líneas es la invocación a la función ***subMethod()***, un método abstracto con visibilidad protegida, cuya implementación es responsabilidad de la subclase ***ConcreteClass***.
De este modo, por cada subclase concreta, la ejecución de la función ***templateMethod()*** respetará siempre la misma estructura, pero con la variación que la implementación ***subMethod()*** pueda ofrecer.

Anexo I

~ Depuración de errores ~

"Depurar es como ser el detective en una película de crimen en la que tú también eres el asesino."

~ Filipe Fortes

Depuración

Bug

Es un defecto en el código de un programa que desencadena en una falla del producto.

Falla

Es la manifestación de un defecto en el código.

Depuradores

Son programas que permiten, como su nombre lo indica, depurar el código fuente. Muchas veces incluyen interfaces gráficas y se encuentran integrados en un entorno de desarrollo. Proveen herramientas que permiten:

- Recorrer paso a paso un programa.
- Detenerse en una línea particular del código, esto se denomina breakpoint o punto de interrupción. Los mismos pueden ser condicionales.
- Inspeccionar el valor de las variables locales y globales
- Observar la secuencia de ejecución del programa (stack trace)

La prueba a ciegas con un depurador no suele ser productiva. Es más útil usarla para descubrir el estado del programa cuando falla, y luego pensar sobre cómo pudo haber ocurrido dicha falla.

Errores fáciles de corregir

La depuración implica razonamiento en retrospectiva. Se debe empezar el análisis desde el punto donde surgió la falla e ir hacia atrás para descubrir las causas. Cuando se tenga una explicación completa, sabrá que arreglar.

Busque patrones familiares

Analice si el error no lo vió antes. Los errores comunes tienen síntomas característicos.

- Errores de conversión de tipos
- Acceso a memoria fuera de los límites
- No inicializar variables

Muchos de estos errores pueden ser detectados por el compilador si se habilitan las verificaciones correspondientes.

Examine los cambios más recientes

Una práctica recomendada es cambiar una cosa a la vez. Si sigue esta regla, es muy probable que el error esté en el nuevo código o haya sido puesto en evidencia por lo último que haya hecho.

Una revisión cuidadosa ayuda a localizar el problema.

En este punto son de gran ayuda los sistemas de control de código (git, svn, etc.)

No cometa dos veces el mismo error.

Después de haber encontrado y corregido un error, revise el código para ver si el mismo error se encuentra en otra parte del código.

Corríjalo ahora, no después

Cuando probando su programa detecte un error, aunque no sea fatal, corríjalo en el momento en que lo detectó, dejarlo para después puede ser más crítico y costoso de corregir.

Obtenga una reconstrucción de la pila de ejecución

Un uso muy común de los depuradores es que permiten hacer un análisis post-mortem. Para esto se obtiene y analiza la pila de ejecución del programa (*stack trace*) la cuál nos brinda la línea de código donde se produjo la falla, además también muestra todas las llamadas a funciones que se fueron sucediendo.

Ejemplo:

```
Program received signal SIGSEGV, Segmentation fault. At ejercicio3.cpp:62

[debug]#0  0x000000000040124c in ejercicio3::ejecutar () at ejercicio3.cpp:62
[debug]#1  0x0000000000401552 in main () at main.cpp:25
[debug]>>>>>>cb_gdb:
```

La pila de ejecución debe leerse desde abajo hacia arriba para seguir el flujo de ejecución del programa.

En este ejemplo vemos que:

- lo primero que se ejecutó es la instrucción que se encuentra en la función main() en la línea 25 del archivo main.cpp
- lo segundo que se ejecutó es la instrucción que se encuentra en la función ejecutar() en la línea 62 del archivo ejercicio3.cpp

El mensaje siguiente nos está indicando que se produjo un error en dicha línea.

Errores difíciles de corregir

Hay ocasiones cuando no se tiene la menor pista acerca del por qué de la falla detectada. A eso nos referimos con *errores difíciles de corregir*. Algunas recomendaciones al respecto son:

Trate de hacer que el error sea reproducible

El primer paso es asegurarse de que puede reproducir el error cuando lo necesite. Diseñe casos de prueba, parámetros y entradas, que provoquen la falla.

Esto ayudará a analizar el problema, encontrar su solución y verificar que no vuelva a ocurrir.

Divide y vencerás

Esta técnica apunta a acotar el fragmento del código donde se genera el error. Para esto suele ser útil saber cuál fue el flujo de ejecución del código, ya que esto nos permitirá descartar algunas partes del mismo. Este accionar se puede repetir hasta dar con la línea que genera el error.

Estudie la numerología de los errores

A veces un patrón en la numerología de los casos de prueba diseñados, que permiten reproducir la falla, ofrece una pista que nos permite enfocar la búsqueda. Analizar los datos de entrada que provocan la falla y su relación puede llevarnos directamente a detectar el error.

Despliegue salidas para situar la búsqueda del error

Cuando no se entiende lo que el programa está haciendo, agregar instrucciones para mostar más información puede ser una forma sencilla y efectiva de averiguarlo. Esta técnica permitirá mejorar su entendimiento sobre el flujo de ejecución del código.

Si va a imprimir el valor de una variable, déle siempre el mismo formato.

Si muestra punteros, hágalo utilizando el formato %x(hexadecimal) o %p(puntero). Esto le ayudará a ver si dos punteros están relacionados.

Escriba un archivo de registro (log)

Este punto es similar al anterior, sólo que enviando la salida a un archivo que contenga la información en un formato fijo. Cuando el programa termina de forma repentina, el registro contendrá la información hasta ese punto.

Asegúrese de vaciar los buffers de E/S para que los registros finales aparezcan en el archivo. En C, una llamada a fflush garantiza que toda la salida se escriba antes de que falle el programa.

Tipos de Errores

A modo de resumen y desde un punto de vista conceptual, podemos agrupar los errores en tres tipos: sintácticos, lógicos y de regresión.

Errores Sintácticos

Son aquellos que se producen cuando el programa viola la sintaxis, es decir las reglas de gramática del lenguaje.

Estos errores suelen ser detectados por el compilador y son fáciles de corregir. Algunos ejemplos de estos errores son:

- Omisión de punto y coma al final de una sentencia.
- Comentarios multilínea sin cerrar. Falta de la secuencia */
- Olvido de doble comillas para cerrar una cadena
- Etc.

Errores Lógicos

Este tipo de error, representa un error del programador en el diseño del algoritmo. Son, por lo general, difíciles de encontrar y aislar. Estos errores no son detectados por el compilador.

Algunos de estos errores puden producirse por:

- Falta de paréntesis en sentencias que calculan algo.
- Falta de inicialización de variables
- Falta de verificación en los datos de entrada.
- Etc.

Ejemplos:

```
// el error está en que el tercer * debería ser un +
double peso = densidad * 5.25 * PI * pow(longitud, 5)/4.0

// esta fórmula no es la correcta
grados_centigrados = grados_farenheit * temperatura_centigrados
```

Errores de Regresión

Son errores que se crean de forma accidental cuando se intenta corregir otro error.

Como regla, y como ya mencionamos antes, antes de compruebe que el error no se produce en otra parte del código y que realmente fue corregido en su totalidad.

Anexo II

~ Pruebas ~

“El mejor momento para encontrar un bug es cuando estás escribiendo el código y no cuando lo estás probando.”

~ Autor desconocido

Pruebas

¿Qué es Testing?

Es verificar que el sistema (o una parte de él) funciona de acuerdo a lo especificado (requisitos funcionales o no funcionales).

¿Para qué hacemos Testing?

¿Qué cualidades del software perseguimos especialmente?

- **Robustez**: distancia al caos/"sensación" de que el código no se rompe.
- **Eficiencia** (en ciclos CPU vs. en tiempo de desarrollo)
- **Corrección**: si hace lo que el cliente necesita (no necesariamente lo que pidió)
- **Consistencia**: el sistema debe comportarse siempre de la misma manera ante un mismo evento y las tareas similares deben poder realizarse siguiendo pasos similares.
- **Completitud**: Un sistema es completo cuando contempla todas las posibles situaciones a darse en la práctica.

¿Qué tiene de malo debuggear?

- Lleva tiempo
- Se pierde el foco en lo que estamos haciendo, solamente quiero que me devuelva 223 como resultado. Pasados los 20/30/40 minutos quedamos envueltos en el código, absorbidos por el monitor y los miles de inspectores de variables que tenemos que tener en nuestra cabeza
- Produce una falsa confianza a la hora de codificar (codifique mal ahora, debuggee después)
- Existe el riesgo de dejar líneas donde hago printf o de borrar líneas de un programa que andaba cuando elimino los printf para que loguee la información

Técnicas de debugging

1. **Fuerza bruta**: "la fuerza del printf". Se prueba una unidad de código con algunos valores de entrada cualquiera hardcodeados y se ve qué resultado arroja imprimiendo el texto por pantalla. Esto trae varios problemas, el principal es que el código queda muy desprolijo. Una vez testeado el programa hay que borrar las líneas de testeo, lo que

puede generar nuevos errores al borrar algo que no corresponde. Otro riesgo es olvidarse de borrar determinada línea de código, lo que puede provocar que al usuario final le aparezcan mensajes que él no debería ver.

2. **Rastreo hacia atrás**: busco donde se produjo el error y miro las condiciones en las que llegó para que diera error.
3. **Eliminar el error**: se comenta la línea donde hay error para ver si se produce un efecto colateral o ese es un error aislado que no trae consecuencias posteriores.

Tipos de testing

Pensando en la forma de ver el sistema, podemos clasificar los tests como:

Tests de caja negra

Se prueba al sistema como un "todo", sin acceso al código fuente, y sin importar cómo se genera la respuesta. Esto puede ser realizado de forma manual (guiándose por un documento o checklist para garantizar que se pueda repetir el test) o también se puede automatizar (escribiendo código que realice las acciones que haría el usuario).

Tests de caja blanca

Se analiza el código desde adentro (+ relacionada con QA). Debe disponerse de acceso a la versión del código que se testeará (no tiene sentido testear código desactualizado).

El objetivo principal es verificar que los flujos de ejecución en el código tengan las salidas esperadas.

En cuanto a la granularidad, los tests pueden ser:

Unitarios

Testean las unidades funcionales más pequeñas (métodos o partes de ellos). El objetivo es tener casos simples para reproducir los errores y fácilmente verificar si están corregidos. Metodologías como TDD los incluyen como parte del desarrollo.

De integración

Su objetivo es verificar la integración de dos o más componentes de un sistema, los que generalmente están desarrollados por equipos o empresas diferentes. Esto generalmente se realiza mediante pruebas de caja negra.

De sistema

Prueban al sistema como una unidad, pensando en la funcionalidad que brinda. Generalmente se utilizan tests de caja negra. Pueden ser manuales o automatizados.

- Smoke tests: Son verificaciones rápidas de funcionalidad.
- Funcionales: Están destinados a verificar casos específicos, flujos de ejecución.

De regresión

se pueden dar en todo nivel de integración. Su objetivo es verificar que los cambios en el código (cambios de funcionalidad, corrección de errores) no “rompan” código que funciona. Para ello generalmente se utilizan tests unitarios y funcionales.

Testing automatizado

Automatizar el proceso de prueba nos permite:

- **separar la lógica de la aplicación de la prueba misma**. De esta forma:
    - al escribir el test de la unidad aparte mantenemos la claridad del software desarrollado. Los cambios realizados en uno no afectan al otro, se puede agregar casos de test sin temer arruinar la funcionalidad de la unidad, y también cambiar el código y luego probarlo con el mismo test de una versión anterior.
- al testear las unidades en un Framework aparte no se mezclan los mensajes de la aplicación con los mensajes relativos al test. De esta forma es más fácil implementar el sistema una vez terminado, simplemente se retiran las unidades de testeo.
- nos brinda la seguridad de que el usuario nunca va a ver un mensaje que era interno para otro programador.
- si los tests son repetibles, las n veces que corra el test, el mismo input termina produciendo siempre el mismo output
- al no haber secuencia, los test corren en forma independiente (tenemos que garantizar ausencia de efectos colaterales)
- como consecuencia de programar la prueba, sólo se invierte tiempo la primera vez que se genera el caso de prueba
- se permite el refactoring: cuando realizamos un cambio en una unidad de código es fácil determinar si todo sigue funcionando correctamente, simplemente se corre nuevamente los test relacionados (uno de los aspectos más importantes en los que influye sobre un equipo de trabajo es el psicológico: la sensación de que el código sigue sin romperse

baja el grado de incertidumbre y reduce el miedo de mejorar). Esto cuenta generalmente para cambios pequeños, si se agrega o quita funcionalidad a la unidad habrá que cambiar el test.

- otra consecuencia del punto anterior es que facilita el código compartido: si al desarrollar software en conjunto escribimos nuestra unidad de testeo y exigimos a todo el que realice un cambio que funcione dicho test, podemos asegurarnos de que siempre va a cumplir con el comportamiento esperado y que un cambio que hace otro no afectará a todo el sistema. Por eso, los tests deben ser lo más claros posibles.
- permite medir el grado de acoplamiento entre los objetos y que reduce el costo de cualquier cambio

Si bien el Unit Testing nos proporciona muchísimas ventajas, es evidente que hay un cierto overhead al escribir las unidades de testeo. Se podría llegar a pensar que al utilizar Unit Testing se consume más tiempo del habitual para realizar un proyecto determinado, sin embargo esto no es cierto. El Unit Testing hace mucho más fácil encontrar los errores, permite el refactoring y permite compartir código... todo esto se traduce en un importante ahorro de tiempo al final del proyecto.

Mock Objects

En programación orientada a objetos, los *mock objects* son objetos que simulan el comportamiento de los objetos reales, de forma controlada. Los mock objects generalmente son programados para verificar el comportamiento de otro objeto, de forma parecida a que los diseñadores de autos utilizan muñecos (crash test dummy) para verificar el comportamiento de un auto durante un accidente.

¿Para qué usar Mock Objects?

En un test unitario, los mock objects pueden simular el comportamiento de un objeto complejo, real (no-mockeado), y por ello son útiles cuando el objeto real es difícil o imposible de incorporar en un test unitario. Si un objeto tiene alguna de las siguientes características, puede ser útil utilizar objetos mock en su lugar:

- Provee valores no controlables (por ejemplo, la fecha actual o la temperatura ambiente) o aleatorios
- Tiene estados difíciles de crear o reproducir (por ejemplo, un error de red)
- Es lento (por ejemplo, una base de datos completa, la cual debe ser inicializada antes del test)
- Aún no existe o su comportamiento puede cambiar

- Debería incluir información y métodos exclusivamente para propósitos de testeo (no para su objetivo real)

Ejemplo

Imagine un programa de reloj que obtiene la hora actual (real) y hace sonar una alarma a un determinado tiempo.

Para testear esto, el test debe esperar hasta que el horario sea el de la alarma para saber si la alarma sonó correctamente. Si se utiliza un mock object en vez del objeto real, puede ser programado para proveer la hora en la que debe sonar la alarma (sea esa o no la hora actual) para poder testear el programa de alarma de forma aislada.

Detalles técnicos: Los mock objects tienen la misma interfaz que los objetos que imitan (polimorfismo), permitiendo a los clientes del objeto desconocer si utilizan el objeto real o mock. Muchos Frameworks de mock objects permiten al programador especificar cuáles (y en qué orden) los métodos serán invocados en un mock object y qué parámetros serán pasados a ellos, así como los valores retornados. Entonces, el comportamiento de un objeto complejo como un socket puede ser imitado por un mock object, permitiéndole al programador descubrir si el objeto que está siendo testeado responde apropiadamente a toda la variedad de estados en los que el objeto imitado pueda estar.

Bibliografía

- Luis Joyanes Aguilar, Ignacio Zahonero Martínez. Programación en C. Segunda Edición. España: McGRAW-HILL/INTERAMERICANA DE ESPAÑA. S.A.U., 2005. ISBN-84: 481-9844-1.
- Brian W. Kernighan, Rob Pike. La práctica de la programación. Pearson Educación. Méjico (2000)
- Debugging with CodeBlocks: http://wiki.codeblocks.org/index.php?title=Debugging_with_Code::Blocks
- Bibliotecas.
    - https://es.wikipedia.org/wiki/Biblioteca_(inform%C3%A1tica)
- Enlace dinámico
    - https://es.wikipedia.org/wiki/Enlace_din%C3%A1mico
- Enlace estático
    - https://es.wikipedia.org/wiki/Enlace_est%C3%A1tico